MASTER IN COMPUTER SCIENCE

Comparative analysis of variability mapping techniques

Leclercq, Marc

Award date:
2011

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la paix, Namur
Faculté d'informatique.
Année académique 2010-2011

Comparative analysis of variability mapping techniques

Leclercq Marc

Mémoire présenté en vue de l'obtention du grade de master en informatique.

Remerciements

Rédiger un mémoire n'est pas une tâche de tout repos et, avant de commencer la présentation de ce document, j'aimerais remercier toutes les personnes qui de près ou de loin m'ont aidé à réaliser cette tâche.

Tout d'abord, je tiens à remercier Messieurs Patrick Heymans et Gilles Perrouin, respectivement promoteur et co-promoteur de ce travail, qui m'ont guidé tout au long de cette année et m'ont permis de mener à bien ce mémoire.

D'une manière plus générale, je remercie également tout le corps professoral des Facultés Universitaires de la Paix à Namur pour son enseignement de valeur.

Aussi, mes remerciements vont à Messieurs Nicolas Guelfi et Benoît Ries, pour leur accueil à l'université du Luxembourg et leur suivi lors de mon stage, lequel m'a apporté une certaine expérience ainsi qu'une base pour la réalisation de ce mémoire.

Egalement, je remercie mes amis qui m'ont supporté tout au long de cette année, et qui m'ont aidé à décompresser lorsque cela était nécessaire.

Enfin, je voudrais aussi adresser mes remerciements à toute ma famille pour son soutien et son aide précieuse au cours de toutes mes années d'études.

Résumé

Le développement de lignes de produits logiciels est un paradigme émergent basé sur la réutilisation d'artefacts dans plusieurs produits d'une même famille, afin de diminuer les coûts et le temps nécessaire au développement logiciel, ainsi que d'améliorer la qualité des produits. L'efficacité de cette approche dépend de la manière avec laquelle elle sait surpasser les coûts additionnels, causés par le besoin de développer l'architecture de la ligne de produits, par les gains encourus grâce à la possibilité de dériver plusieurs produits de la même ligne d'une manière plus aisée. Ces gains peuvent être augmentés par l'automatisation du processus de dérivation de produits. Dans le cas de lignes de produits logiciels utilisant des modèles de features pour représenter les différents choix possibles, et des modèles architecturaux pour représenter les détails architecturaux de la ligne de produit, une telle automatisation requiert de spécifier les liens existants entre ces deux types de modèles, ce qui est également appelé en anglais « the variability mapping problem ». Si différentes approches à ce sujet existent, le domaine manque cependant de comparaisons sur leurs avantages et inconvénients. Dès lors, dans ce mémoire universitaire, nous établissons une comparaison de plusieurs des principales approches de ce problème, premièrement sur base de la littérature et deuxièmement sur base d'expérimentations pratiques, dans le but de guider les ingénieurs du domaine et de fournir des pistes d'améliorations pour ces approches et outils.

Abstract

Software product line engineering is an emerging paradigm that is based on reusing artifacts for some products of a same family in order to decrease costs and time efforts required by software engineering, and to increase the quality of the products. The efficiency of this paradigm depends on how much the added costs of developing the product line architecture are outweighed by the gains of being allowed to derive multiple products from the product line in an easier manner. These gains can be increased by automating the derivation of products. In software product lines using feature models to represent the various choices, and architectural models to describe the architectural details of the product line, the automation of product derivation requires to specify the links between both models, which is also called the "variability mapping problem". If different approaches to this problem exist, comparisons of their relative benefits and drawbacks are currently missing. Therefore, in this master's thesis, we make a comparison of several of the most-known techniques, based firstly on literature and secondly on practical experimentations, in order to assist software product line engineers and outline future directions regarding the development of such approaches and tools.

Table of contents

Table of figures

List of tables

1 Introduction[1]

Until the 1990's, software solutions were relatively monolithic and had all possible features that one might ever need. In fact, software was produced for a single purpose, for a single type of customer, and changes in the requirements implied a lot of creating and editing source code. But, in the other hand, the complexity of software systems grew over time due to the evolution of the market, such as with the development of distributed systems, and so have the expectations of the customers in quality, and the need to produce software in a time and cost effective manner.

Therefore, the traditional software development approach showed its limits for handling these expectations, as it implied too much efforts, time and money to develop software that was not always meeting the quality requirements.

Hence, another software development paradigm, said to handle these issues in a more effective manner, emerged rapidly. This paradigm, the software product lines paradigm, is based on the fact that actual software is mainly seen as redevelopment, as most products have been built before at some level [IKJ10]. It was first introduced by Parnas [Par76] and emerged when the concept of reusing software artifacts began to be seen as the way to solve quality, time-to-market and cost issues of software development. With software product lines, numerous base software artifacts are developed, and then re-used in multiple products depending on design decisions regarding the products' requirements, which also satisfy the need for mass customization of software, and avoid having to re-develop each new type of product entirely.

The derivation of products from the product line is key to the efficiency of the product line approach, as it is where additional efforts to implement artifacts of the product line instead of implementing a single product should be outweighed by the gains in time, cost and quality provided by reusability. This derivation thus has to be as much automated as possible in order to maximize the gains. However, studies [DSB041] have shown that product derivation is often carried out manually, which makes it more difficult, time-consuming and error prone [BLT08]. This fact underlines the need to bring more automation to this process.

In order to derive products from a product line, we first have to select the needed features of the product, from the features provided by the product line, which are represented using feature diagrams. Feature diagrams basically are tree representations of features that define the common and variable characteristics of a product line. Then the product has to be constructed using the artifacts related to the selected features. That phase, often carried out manually, can be automated if we succeed to specify formally the links between those features and artifacts. When artifacts are represented using diagrams such as class diagrams, the mapping between features and elements of these models allows to generate architectural model of the derived products, which can then be the input for model driven engineering techniques that will generate the final product.

The problem of mapping features to model elements is also called the "variability mapping problem". In our work, we will analyze different approaches that handle variability

[1] This master's thesis is partially based on the work done during an internship at the University of Luxembourg. A small report of this internship can be found in annex C.

mappings between features and UML class diagrams. We deliberately exclude approaches considering other variability models. Moreover, the domain of software product lines is not completely mature yet, and for example new approaches to the variability mapping problem are still researched and created. For most of these approaches, they haven't been compared to other approaches, and their respective advantages and drawbacks are not well known. Therefore, the objective of our work is to compare some of the renowned existing approaches (comparing all of them is not feasible) and try to highlight their advantages and disadvantages.

This comparative analysis of variability mapping techniques will start with a presentation of its context. We will introduce the concepts of software product lines, software product line engineering, feature modeling, UML diagrams, variability mapping, and some other concepts that we will use within this master's thesis. Then, we will introduce the different approaches to the problem that we will study. We will then be able to make a comparison of these approaches in literature-based point of view. To do so, we will first present our literature-based comparison criteria, which are (explicit) expressiveness, usability, adaptability and maturity, and then apply these criteria to each approach before concluding this first comparison. Thereafter, we will make a second comparison, based on an empirical approach of the problem. In fact, we will apply a small case study (which consists of implementing a product line) to each approach after having defined this case study, and our empirical comparison criteria, which are expressiveness, usability and performance. Finally, we draw the conclusion of this work.

2 Context

In this section we will introduce the different concepts, approaches and theories that are related to our work, the comparison of variability mapping approaches.

2.1 Software product lines

The concept of software product line ("SPL") is based on the concept of reuse, which has been the long standing notion to solve the cost, quality and time-to-market issues associated with development of software applications since the 1960's [DSB041]. A major addition in this domain was in fact the introduction of the software product families approach by Parnas [Par76], which drew more and more the attention of the software engineering community when software begun to be integrated massively in families of hardware products [Per07]. In fact, the new trend in software engineering is the need to develop multiple, similar software products instead of just a single individual product [BeDa06] because products have to be adapted to different markets (because of different languages, laws, etc.) and also because software is becoming more and more complex, so re-developing each product from scratch requires a lot of time and money. Also, it showed that, in specific domains, the actual development of software is mainly seen as redevelopment, since most products have been built before at some level [HeTr03].

Therefore, the main idea of a software product line is the exploitation of similarities between products of a same family by reusing some of their shared assets (an asset is defined by Withey [Whi96] as "a description of a partial solution (such as a component or design documents) or knowledge (such as a requirements database or test procedures) "), instead of re-implementing the assets for each product. This has proved to substantially decrease costs and time-to-market, and increase the quality of their software products [vdL02] and so the efforts needed to develop the product line are outweighed by the benefits of reusing artifacts that are already tested and secured to define, design, build and maintain the products of a same family.

A software product line thus describes products that belong to the same domain and that share common artifacts (we say that these products belong to the "scope" of the product line). This idea is captured by the definition given by Clements and Northrop [ClNo01] : "A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way".

Re-using artifacts in a software product line provides various benefits. In [ClNo03] Clements and Northrop defined these various benefits as organizational benefits, such as a better domain comprehension or the customer's trust, software engineering benefits, such as the reusability of requirements and components, a better analysis of the requirements and avoiding redundancy, and business benefits, such as the gain of time and money thanks to a better efficiency in the development process.

A product line contains different assets that can be shared between every product, or specific to one or multiple products. Those assets define the commonalities and variabilities of the product line. The commonalities of a product line are assumptions that are true for every specific product that is in the scope of the product line, while variabilities are assumptions

about how those products differ [IKJ10]. For example, for the most known product line example, the cellular phone product line [MaHe05], the commonalities contain what every phone in this line must possess, such as the ability to make phone calls, and the variabilities define differences such as the size of the screen, the available languages, the size of the memory, etc. In a product line, the variability is usually defined using "variation points" and "variants"[2]. A variation point is a place in the design or implementation that identifies locations at which variation will occur [Jac97] and is composed of variants that represents the different alternatives that are possible for this point (for example, the different screen sizes). The commonalities and variabilities of a product line are often represented by feature models (see "feature modeling" below). In [HeTr03], the variability is decomposed into different types:

- The variability in function defines optional functions that may appear in some products
- The variability in data defines options for the way of representing data with different data structures,
- The variability in control flow defines options for the patterns of interaction
- The variability in technology defines different options for the technologies used for the platform
- The variability in product quality goals defines different levels of goal satisfaction (such as performance or security)
- The variability in product environment defines different requirements to satisfy depending on the environment.

In a product line, a specific product is defined in a "configuration" (also called "variant"). It is defined in [DSB041] as "an arrangement of components and associated options and settings that partially or completely implements a software product", and it thus contains the commonalities of the product line, as well as the variabilities assets required by the product. This configuration can then be used for the process that creates the product by composing reusable components, and which is called "product derivation". Product derivation is defined in [DSB041] as "the complete process of constructing a product from product family software assets". The product derivation process is defined in the next section.

The different assets of the product line are contained in the product line architecture. This architecture contains a set of architectural decisions, a set of reusable components and eventually a library of optional components corresponding to specific clients' requirements [HeTr03]. The SPL architecture (SPLA) is also defined in [IEE00] as " the higher level structure that is shared by the product family members and that denotes the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution" . Each product family member derives its architecture from this overall structure.

The process of developing a software product line is called software product line engineering (SPLE) and is defined in the next section.

[2] In this master's thesis, we will call these variants "options" because variant is also used to define a specific product configuration.

2.2 Software product line engineering

Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization [PBvdL05]. Platforms are collections of reusable artefacts that are used by combining them in order to create specific applications. The aim of the mass customization is the "large scale production of goods tailored to individual customer needs [PBvdL05]". Software product line engineering ("SPLE") is becoming more and more important because of market changes: customers now want more customized products than before. As said in [vdML04], "the development of software-intensive systems shifts more and more from single product to software product line development", pointing out the growing importance of this paradigm. Reusability of artefacts is a key fact of SPLE, as it is frequently the case that software development is actually a redevelopment process because many products have been partially built before [Ist10].

The process of SPLE as defined by Klaus Pohl is described by Figure 1.

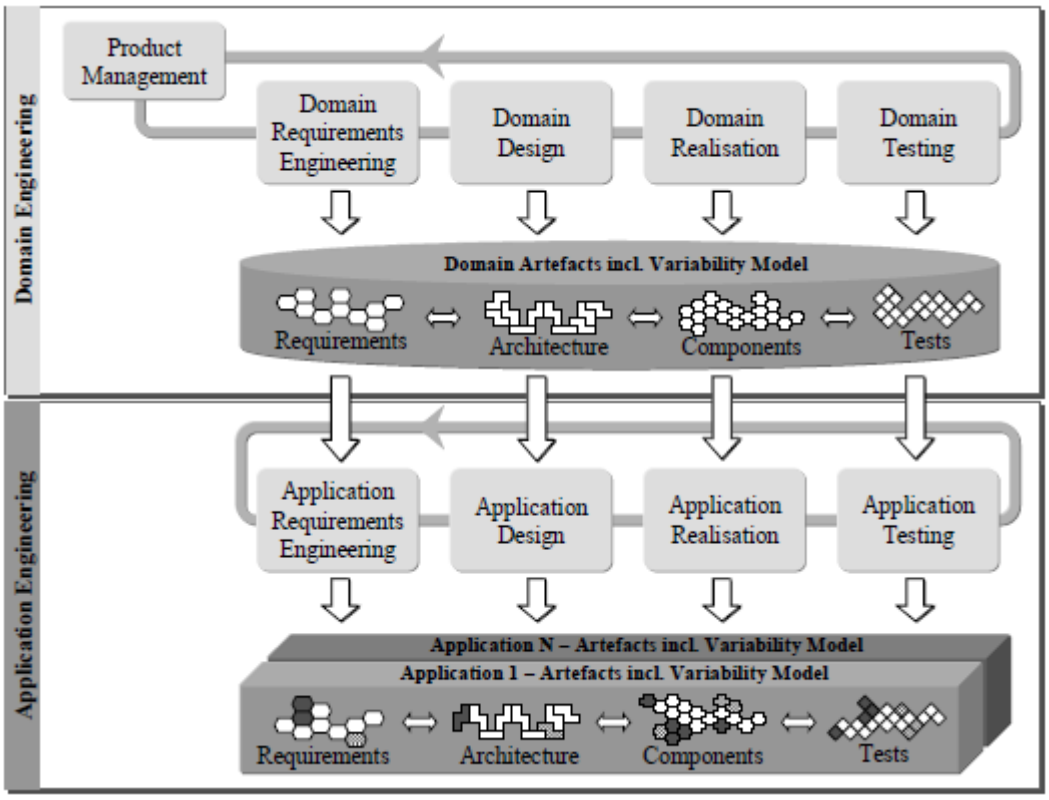


Figure 1 : process of SPLE [PBvdL05**]**

This process is divided in two sub processes: domain engineering and application engineering.

Domain engineering is the process where the commonalities and variabilities of the SPL are defined and realised via the sub-processes "Domain requirement engineering",

Domain design", "Domain realisation", domain testing" and "product management". It is thus in this process that we define and implement the core assets of the product line.

- *Product management* is about defining the scope of the product line, i.e. the boundaries that define if a product belongs to the product line or not.
- *Domain requirement engineering* cares about defining customer visible variability by developing domain requirements artefacts (use cases, feature diagram, etc.)
- *Domain design* defines the "technical" variability of the SPL, thus the variability that users cannot see, and has to define the mechanisms that supports the variability
- *Domain Realisation* is the process where the architecture artefacts are designed in details and implemented.
- *Domain testing* cares about developing test artefacts like test cases and verifying and validating each artefact separately with the test artefacts.

Application engineering is the process in which product variants (i.e. applications derived from the product line) are developed using reusable artefacts (all artefacts that belong to the commonalities plus a selection from the artefacts that belong to the variabilities of the product line) from the platform and other artefacts specifically made for the variant. The selection of "variability" artefacts is obtained by specifying which features we want for that variant while respecting restrictions specified about the relation of the features (for example, two features can exclude each other). Application engineering contains the sub-processes "application requirements engineering", "application design", "application realisation" and "application testing".

- *Application requirements engineering* is about defining the requirements of the product variants by defining the customer visible variability bindings. The application requirements are thus derived from the requirements artefacts from the platform that are bound to the variant.
- *Application design* defines the "technical" variability bindings and derives the variant architecture from all the previously specified variability bindings.
- *Application realisation*'s aim is to create a configuration of the components of the variant architecture in order to produce a working application, the product variant.
- *Application testing* derives test artefacts from variability bindings and uses them to verify and validate the created application.

Product derivation

Product derivation is a process applied during application engineering [BLT08] which consists of deriving a product from the product line. It is defined in [DSB041] as " the complete process of constructing a product from Software Product Line (SPL) core assets".

Product derivation is key to SPL approaches [PKGJ08]. However, a quite recent study [DSB041] shows that the product derivation is often carried out manually (the problem is also underlined by Haugen in [Hau04]), and thus makes it more difficult, time-consuming and error

prone [BLT08]. It also decreases the advantage of using Software product line engineering rather than standard product development.

Therefore, there is a need of automating the product derivation process and to provide tool support for it. If the feature configuration of a product (i.e. selecting the features for a product) requires to be made by human, it can be already supported by tools that provide the ability of checking the validity of the selection of features depending on different constraints. This configuration can then be used by a tool that processes the product derivation automatically, instead of doing it manually [BLT08].

Several automated product derivation approaches have been proposed, most of them using model-driven techniques to derive products according to choices made by product engineers on the basis of a decision model [PKGJ08]. For example, in [BLT08], they propose an approach supported by a research tool that uses model transformations and AspectJ[3] mechanisms to perform the assembly of the particular products. Other examples are of course the different approaches that we will compare in this wok, excepted FLOCOSGPL, which is not an automated approach.

A product derivation approach can be of two types: configuration or transformation.

- Product Configuration is based on the parameterization of SPL core assets rather than focusing on how individual products can be obtained. It is based on making a selection of features and then the automatic assembly of reusable assets by a "configurator" tool and thanks to a decision model contains the necessary constraints and traceability information in order for the configuration tool to make the right decision [Per07].
- Product transformation is based on how individual products can be obtained from the product line. It is based on techniques such as MDE (see further) that allow to transform the model and its core assets directly, but it also requires a fully defined decision model.

Such as stated in [PKGJ08], product derivation often encounters problems when facing the clients' requirements, which can introduce new features that were not expected. In order to handle the derivation of products similar to those of the product line but not fully implementable with the current product line assets, [DSB041] (and [DSB04]) proposes a "generic product derivation process", shown in Figure 2 . This process contains the following steps:

- The initial phase. In this phase we create a first configuration from the product line assets. The input is a set of requirements, which are likely to be not fully satisfied with the product line current assets. The initial phase can be done either by "assembly" or by "configuration selection".
    - Assembly consists of assembling a subset of the core assets. There are two assembly approaches: construction or generation.
        - Construction consists of creating the initial configuration from the product family architecture and shared components

[3] AspectJ is a Java extension that supports Aspect Oriented Programming (see further)

by deriving the architecture, selecting the closest matching components and setting the parameters.

- Generation consists of modeling the shared assets instead of implementing them in source code, then selecting a subset of these assets to create an "overall model" and then generating an initial implementation from this model.

- Configuration consists of selecting a "closest matching existing configuration" of the assets, which can be an old one created for another product, or a reference configuration that was created for being reused in this case, and contains a base configuration. Then, we re-derive the architecture, and add and remove some components of the product line, and then reset the parameters.

- Finally, the first configuration is validated.
- The iteration phase. In this phase we apply a modification of the configuration for each iteration, until the product sufficiently implements the imposed requirements. So, if the initial configuration doesn't sufficiently implement the requirements, we apply modifications iteratively. These iterations are made of a modification phase and a validation phase. For the modifications, we can only have to reset some parameters, or we can have to apply some adaptations, which need to rederive the architecture and reselect some components. Those adaptations can be "product specific adaptations", meaning that we create new functionalities in a new product specific asset, or "reactive (or also proactive) evolution", which consists of evolving already existing assets in order to satisfy new requirements.
- Finally, we the validation is passed successfully, the new product is "ready".

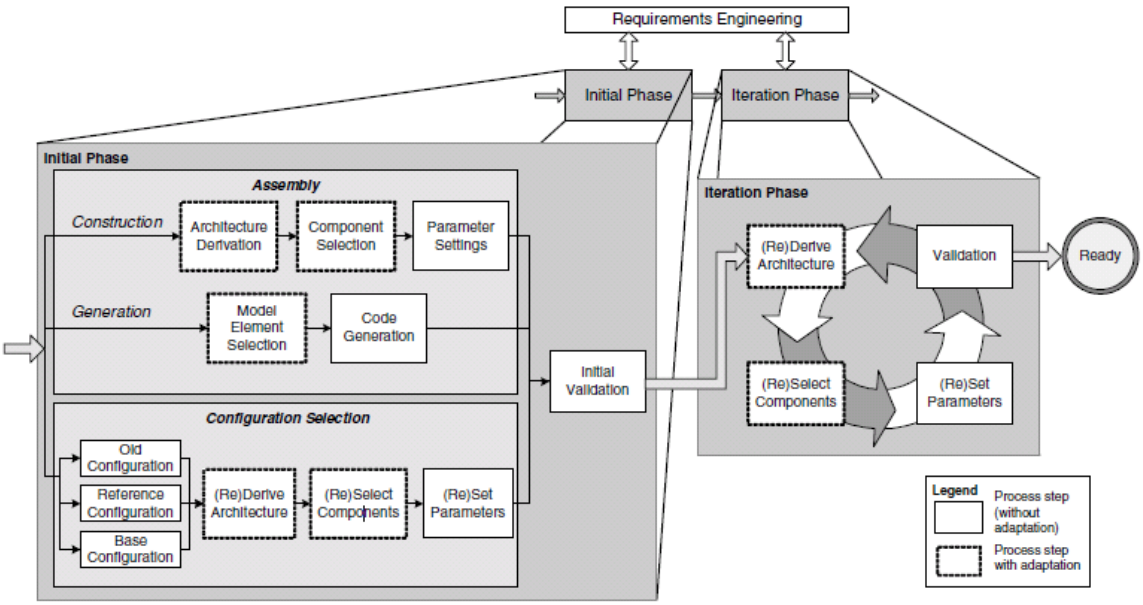


Figure 2 : Generic product derivation process [DSB041**]**

2.3 Feature modeling

An important part of the product line paradigm is the modeling and management of variabilities and commonalities at requirements, architecture, components and test levels, and allowing the user to make decisions concerning this variability [IKJ10]. This is often made using feature models.

A feature model is composed of features. A feature is an end-user visible characteristic of a system [Kan90]. A feature usually refers to requirements but can also represent domain properties, specifications and design, which can lead to confusion about what the diagram is describing [IKJ10]. According to [BeDa06], the meaning of a feature diagram needs to be decided for each Product Line. In [SHTB07] Schobbens and Heymans distinguish the use of features in the two SPLE phases: during domain engineering, a feature is used as a unit of evolution that adapt the family to an optional requirement. During application engineering, the feature usage is to be selected (or not) in a product configuration to define it. These products are composed of a lot of features, which brings the need of having well-identified relations between those features. So, the usage of a feature is to represent the variability of the product line as well as constructing new products and identifying constraints of the product line.

In general, features are organized in a tree, or in a directed acyclic graph [MeHe07]. This tree thus depicts the variability of the product line. It contains variation points, which are"features that have at least one direct variable subfeature" [Per07].

There are multiple types of features: a mandatory feature is a feature that must be selected in every variant, as opposed to an optional feature. An alternative feature is a feature that is part of a feature group where one and only one feature must be selected. An OR feature is a feature that is part of a feature group where at least one feature has to be selected [BeDa06]. Finally, a AND feature belongs to a feature group where every feature has to be selected. Features can also have cardinalities, such as in [CHE05] , which can be considered as another way to express mandatory or optional features, as well as the type of a feature group, such as in [RBSP02]. Czarnecki et al. [Krz02] also introduced feature attributes as a way to represent a choice of a value from a large or infinite domain such as integers or strings [Per07]. Features also have relations between them, such as requires or conflicts [BeDa06], or also relations of generalization [vGBS01]. Features can also be organized in layers and then belong to the category of context features, representation features or operational features [GFdA98].

Several feature modeling languages exists, and no single standard has been agreed for the graphical notation of feature models, even though the graphical form of the FODA method is common [BeDa06]. So here we will present the most common feature model languages, such as analysed in [IKJ10] and also in [MeHe07]. Figure 3 [IKJ10] presents a comparison of those languages' graphical notations.

FODA

Feature-Oriented Domain Analysis (FODA) was introduced in 1990 [KLD02]. A part of FODA is OFT, the first ever feature modeling language [Kan90]. According to [IKJ10], this notation has the advantage of being clear and easy to understand, but lack of expressiveness as

it cannot express relationships between variants or explicitly represent variation points and generalization/specialization relationships.

FORM

Feature-Oriented Reuse Method (FORM) [Kan98] is an extension of FODA which integrates "implemented by", "composed of" and "generalization" links to it. It also offers to categorize features into layers (capability layer, operating environment layer, domain technology layer and implementation technique layer) and features are depicted into boxes [IKJ10].

FeatuRSEB

FeatuRSEB [GFdA98] is a combination of FODA and the Reuse-Driven Software Engineering Business (RSEB) method. RSEB is a use case driven reuse process, where variability is captured by structuring use cases with explicit variation points and variants [Ist10]. It introduced graphical notations for constraints (arrows) and is also a directed graph.

Van Gurp et al.

The language proposed by Van Gurp et al. [vGBS01] extends FeatuRSEB to deal with bnding times, indicating when features can be selected, and external features, which are technical possibilities offered by the target platform of the system [MeHe07].

Generative Programming

Generative Programming [CHE04] is an approach that makes possible the automatic program generation. Czarnecki and Eisenecker adapted FODA FDs in the context of Generative Programming. By adding OR features and textual constraints to make possible the automatic program generation.

Riebish et al.

The language proposed by Riebish et al. [RBSP02] extends previous FD languages focusing on the cardinalities. In fact, AND and OR decomposition are replaced by the expression of cardinalities.

PLUSS

The Product Line Use case modelling for System and Software engineering (PLUSS) approach [EBB05] is an extension of FeatuRSEB that combines feature diagrams and use cases to express a high level view of the product line. It allows the graphical representation of constraints and introduces "single adapters" and "multiple adapters" in order to represent XOR and OR decompositions.

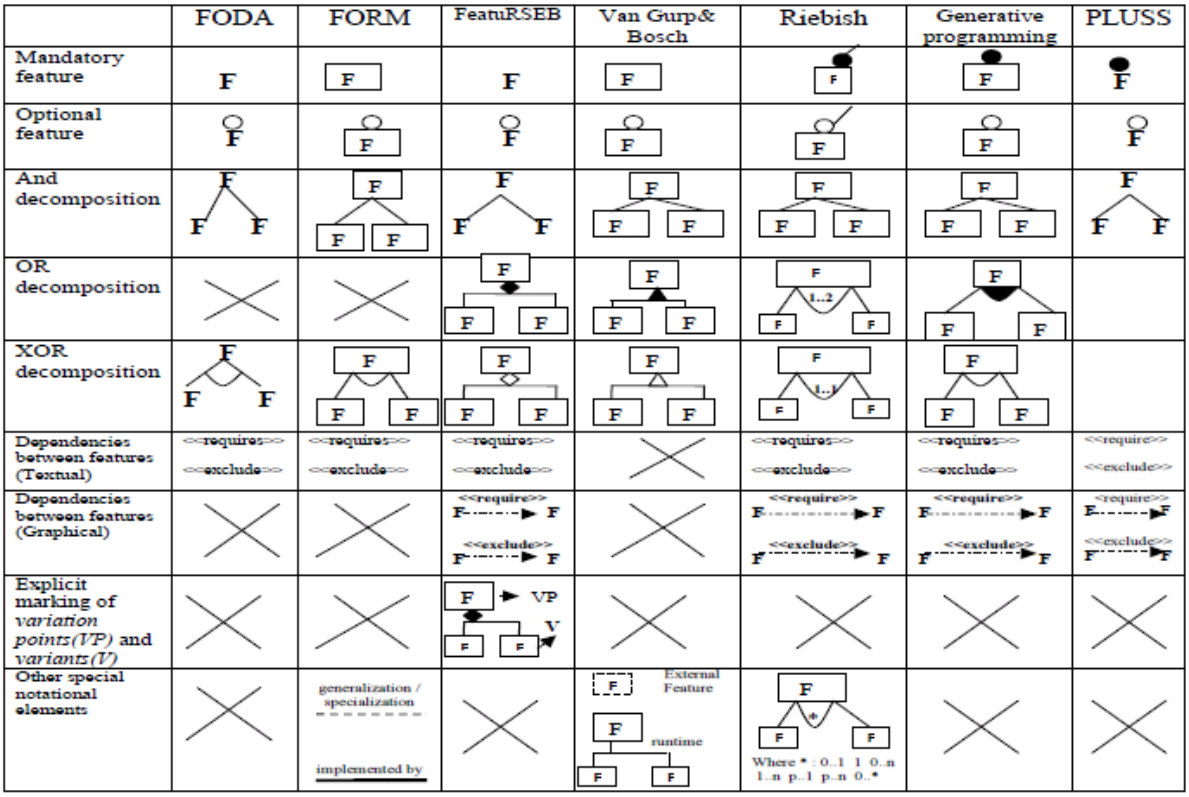


Figure 3: Graphical notation of the most common feature modelling languages [Ist10**]**

2.4 UML

In order to describe how the SPL is realized and model the solution space of the SPL, we need other models han feature models. To do so, we can for example use UML models, such as UML component models or UML class models. In this work, we will use UML Class models.

UML [OMG05], as known as Unified Modelling language, is the result of a standardization effort in object-oriented modeling methods that started in 1994. The standard was created and is managed by the Object Management Group (OMG). It is used to specify, create and visualize the artifacts of a software system.

UML is based on a multiple layer approach. In fact, it has a four-layer object-oriented metamodelling hierarchy [OMG05].

- The first layer, called M0, is the layer where elements are objects as present in the memory of a computer running the system.
- The second layer, called M1, is the layer composed of the models that are created by users.
- The third layer, called M2, is the layer that defines the UML meta-model.
- The fourth layer, called M3 or MOF (meta-object facility) , is the “meta-metamodel” and is a model manipulation framework designed to define modeling languages.

2.4.1 UML class diagram

An UML class diagram is diagram defined by UML that describes the structure of a system by showing the system's classes, their interrelations, and their attributes and operations. Classes represent types of objects in the object-oriented paradigm. Class diagrams are use for a "wide variety of purposes, including both conceptual/domain modeling and detailed design modeling" [Agi11].

An example of UML class diagram, which we will use for our case study in our empirical comparison can be found in Annex A.

2.5 Variability mapping

Feature models are used to express the variabilities of a product line. However, they do no express how specific features are realized, and so they cannot express how products are realized within the product line. To describe how the SPL is realized and model the solution space of the SPL, we use other models such as UML class diagrams. Therefore, it leads to a mapping problem: we need to identify which elements of the solution space models are linked to the features of the feature models in order to be able to derive products automatically from a feature selection. The activity of specifying the links between those is called variability mapping [Hei09].

Variability mapping is "the activity of expressing explicitly the relationship between features and model elements" [Hei09]. It bridges the gap between the description of the variability of the product line (with the feature models) and the architectural elements of the product line by specifying the links between them, which can be used by tools in order to provide an automated product derivation. In fact, the user of such a tool will only be asked to select the features that he wants to create the product, while the links (along with other models or processes) will allow to create the final product containing the good artefacts.

In order to manage the variability of a product line, we can use three techniques: negative variability, positive variability and parameterization of model elements. Positive variability makes use of a model that contains core assets of a product line (the "base system"), and then allows the user to add new elements when selecting additional features. Whereas negative variability uses a model of the complete product line, which thus contains all possible model elements in the product line domain, and then allows the user to remove model elements that are not mapped to any feature selected by the user. With negative variability, we thus have to possess a full model as input of the process. Therefore, if the software product line contains a lot of architectural elements, the model will be huge and then not very readable for the user. With positive variability, we avoid this problem by separating model elements for each different product into different models. But, at the same time, this means that there can potentially be a large number of model fragments describing the SPL as a whole, making it difficult to get an easy overview of the SPL and to spot the interaction between features and elements. Finally, the parameterization of model elements consists of creating rules that allow to modify the values of attributes, of the name of the class, etc.

Figure 4 presents an overview of the variability mapping approaches that are available for static software product lines (static SPL's are product lines where the variability is

resolved when constructing a variant, and not at runtime). Those techniques can be declarative or operational.

Declarative variability mapping techniques use declaration techniques such as annotations on a model in order to describe what changes are needed in the target models when selecting/deselecting a feature. They do not allow specifying how the models have to be modified to apply the changes, but these mechanisms are encapsulated in the tools [Hei09].

The declarative techniques are decomposed into the techniques that use positive variability and the techniques that use negative variability. The negative variability techniques can use direct annotations, i.e. they add annotations on the target models themselves, or use a separate annotation model in order to represent the mapping, such as the "mapping models" in FeatureMapper. Parameterisation of model elements consists of modifying the architectural elements based on the selection of features.

Unlike declarative techniques, operational techniques allow the user to specify how targets model must be modified depending on a selection of features.

The operational techniques are generic model transformations, aspect-oriented modelling and customized model transformations. Generic model transformations use a model transformation language that allows the user to generate a model depending on the input model (here, a feature model), and the transformations that the user specified and that are applied when a certain feature/ type of feature is contained in the input model. Aspect-oriented modelling is a modelling technique that provides means for modularizing crosscutting concerns, encapsulated as aspects [GrVo09]. Then, an AOM mechanism, often called "weaving", support the composition of different aspects in order to develop specific products (see section 2.8). Customised model transformations allow the user to define a model transformation language himself. Then he can use this language to specify transformations in the same way than with a generic model transformation language.

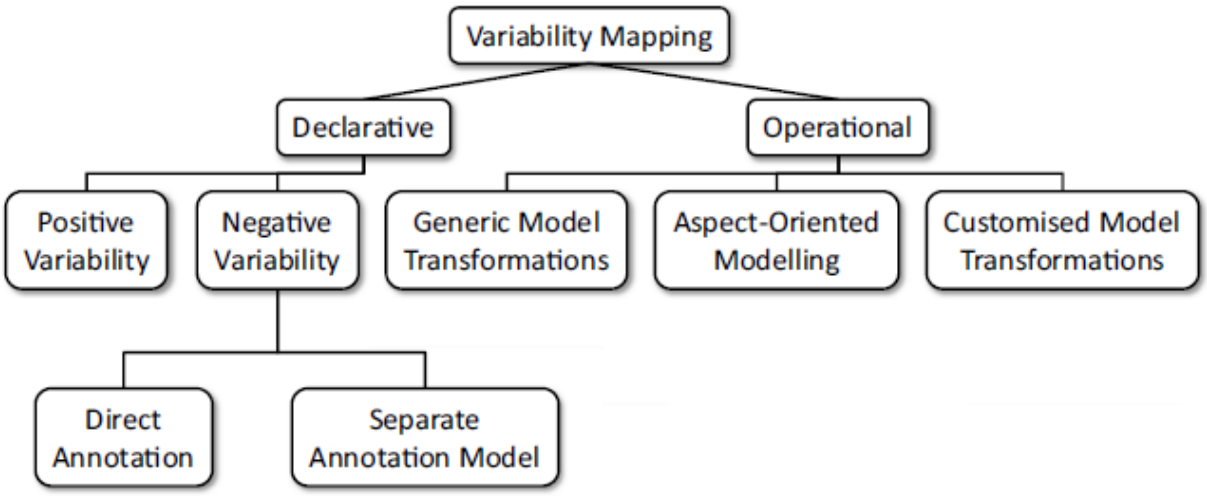


Figure 4: Overview of variability mapping approaches [Hei09**]**

In this master's thesis, we will make a comparison of approaches to the variability mapping problem that can make the mapping between features of the software product line (independently from how they are modelled) and elements of a solution space model expressed in a UML class model, in order to derive an architectural model of the derived product. Therefore, approaches that derive products directly from a feature configuration without using a UML class diagram for the solution space, such as Pure::variants and its family model [Pur09], does not belong to the scope of our work.

2.6 Crisis management systems

In this thesis, we evaluate different variability mapping approaches. The empirical evaluation will be supported by our case study consisting of a Crisis Management System, named REACT.

A crisis is defined by [SSU98] as "a rare event that comes surprisingly, threatening an organization and forcing a short decision time". The need to handle those crises is becoming more and more important as crises are occurring more frequently and are more difficult to handle because they change more rapidly [RoLa02]. Crises are characterized by the uncertainty over what is happening, with large gaps between what is really happening and what the people that supervise the handling of the crisis think it is happening [NES07]. Those people have thus difficulties to make the good decisions, especially when they lack of tool support to help them.

Also, the traditional crisis management approaches (without tool support) encounters problems for having rapid access of vital information, problems for training people and structure problems [RoLa02]. So, the need for a Crisis Management System has grown over time [KGM10] to respond to those problems.

Crises are decomposed into two types: smoldering crises and sudden crises. A sudden crisis is an "unexpected event in which an organization has virtually no control and perceived limited fault or responsibility" [Las09]. Smoldering crises are "events that start out as a small internal problem within a firm, become public to stakeholders, and, over time, escalate to crisis status as a result of inattention by management" [Las09].

Sudden crises, which are often the crises handled by the crisis management systems, can be also categorized by their nature. In fact, a crisis can range from natural disasters such as floods or earthquakes to terrorist attacks, accidents (explosions, crashes),etc. [NES07]

So, a crisis management system is made to handle those kinds of crises. According to [KGM10], the general objectives of a CMS include the following:

- To help in the coordination and handling of a crisis;
- To ensure that an abnormal or catastrophic situation does not go out of hand;
- To minimize the crisis by handling the situation using limited resources;
- To allocate and manage resources in an effective manner;

- To identify, create, and execute missions in order to manage the crisis;
- To archive the crisis information to allow future analysis

It has to provide to coordinators and observers a way of managing the complexity of the crisis by providing rapid and efficient access to data, as well as efficient means of communication among the different stakeholders, which can be, along with coordinators and observers, policemen, firemen, doctors, governmental services, etc.

The process of handling a crisis with a CMS must handle the authentication of users, the inititation of a crisis state based on observations, defining and attributing missions to resolve the crisis, but also be able to archive the data of the crisis in order to benefit from the knowledge of previous experiences [KGM10].

In [Lec11] we propose the following definition of a CMS : " A crisis management system is a system that is made to handle a specific crisis in an effective manner. It has to facilitate coordination of activities and information flow between organizations that are asked to handle the crisis (for example, hospitals, police departments,..), with limited resources, and to accumulate some expertise in the domain by storing informational and result data in a database. It is thus composed of humans and machines like communication systems".

Because of the large scope of possibilities of a CMS, applying a SPLE approach to the development of such a product line is very interesting. For example, the LASSY laboratory of the University of Luxembourg created the crisis management system product line REACT, which we will use for our empirical case study. The REACT product line provides core assets that are used to derive crisis management products.
It was made to allow the creation of specific applications that can handle one special sudden crisis like a flood, a storm, a car crash, a chemical plant explosion, etc. This product line has been modelled in a feature diagram in [KGM10] (Figure 5) and a more detailed feature diagram of REACT can be found in [Lec11] and in Annex B. Basically, the REACT product line can handle different sudden crises types, such as natural disasters or major accidents, different sizes of areas, different IT options such as surveillance systems and different means of communications such as PDA's or talkie walkies. It also implies internal and external resources, which can both be human or material, such as coordinators, observers and system admins for internal resources and police, army, firemen for external resources. A crisis management system from REACT can assign different types of missions, for example "observe", "transport" or "repair". Finally, it can imply witnesses or victims.

These features represent the commonalities and variabilities of REACT. The implementation of the product line mainly consists of Java Artefacts, and Annex B shows a class diagram of a part of these artefacts.

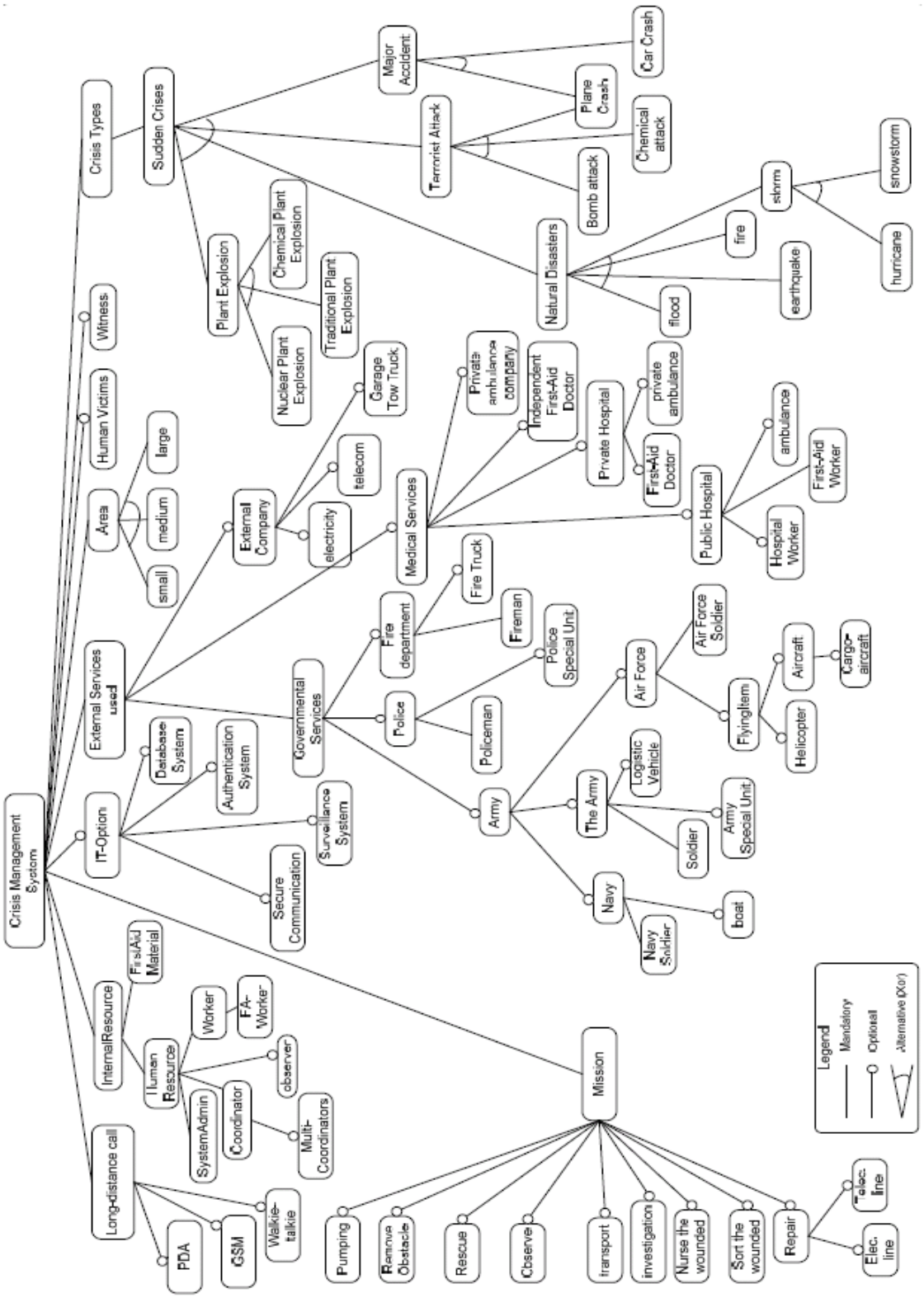


Figure 5: feature model of the REACT platform [KGM10**]**

2.7 MDE

Model driven software development is a methodology that "is based on standardised models (such as UML models) that are refined, transformed and eventually translated into executable code using code generators" [HJSW08]. It can be used in the context of software product line engineering to automate the product derivation thanks to the model transformations [Hau04]. Also, models help to understand the product line , reason and communicate about the product line requirements, which gives it, along with the ability to transform models, a "a prominent role to play in product-line engineering to define their core assets and support product derivation" [PKGJ08].

A model transformation is defined using a model characterizing the different transformations rules required to transform one or more source models into one or more target models. A transformation rule, or transformation operation, describes how one or more elements of a source model have to be transformed for the derivation, and these rules can then be processed automatically by a tool to perform product derivations.

In the case of product derivation, most of the automated approaches use MDE techniques to automate the derivation. For example, the approach of Czarnecki [CzAn05], that we will compare in this work with other variability mapping approaches, uses MDE techniques to derive products.

2.8 Aspect Oriented Programming

Aspect Oriented programming is a software development paradigm that focuses modularisation in order to isolate secondary functions, also called "cross-cutting concerns". It come from the idea of "separation of concerns" introduced by Dijkstra and Parnas [Scha02], where the main idea the identification of different concerns in software development and their separation by encapsulating them in appropriate modules or parts of the software.

The aim of AOP is to implement these concerns in one aspect only, instead of having its implementation scattered in different places. These techniques, and Aspect oriented Modelling among others, can be used for modularizing feature implementation [BLT08], and thus can be interting in the scope of software product lines. For example, the approach of CVL [Fle09], which will be part of the comparison in this work, is based on Aspect Oriented Modelling (AOM). AOM is thus about separating different concerns in the modeling level, and the different aspects of those models can then be composed into a representation of the complete system [Hei09].

AOP techniques thus implements concerns into one aspect that encapsulates the concern. Aspects can be combined at specific points of the system that are called join points [Kic97] . Those join points are interpreted by a special compiler called the aspect weaver, which brings a composition mechanism to coordinate aspects to other modules of the system.

2.9 Feature Oriented Programming

Model driven engineering can bring automated product derivation to a product line with high-level modelling and code generation from the models. Feature Oriented Programming [BSR04] (FOP) is a complementary approach to MDE that allows building software product lines by focusing on the features and their combination. For example, in [TBD06], they combine MDE and FOP approaches to make product derivations.

Feature oriented Programming comes from the late 1980's, where the design of programs was made in layers, with each layer containing one functionality. There was then the need to have a language to express such design, this language appearing to be elementary algebra. Then, the idea of layers was generalized to features.

In Feature Oriented Programming, features (or "feature modules" are not only a representation of the variability, but also the building blocks of the program (a feature is thus also an architectural artifact). So, each feature may include any number of artifacts. In fact, FOP sees features as design and implementation entities, which means features are designed and implemented as program refinements [BLT08]. For the derivation of products, feature Oriented Programming makes use of algebraic techniques to combine features [TBD07].

2.10 Traceability

In a software product line, a methodological approach is needed to harvest all of its benefits. Therefore, according to [Anq08], the variability of the product line needs to be managed in an appropriate and consistent way across the different SPLE processes, and thus requires, among others, to provide traceability.

Traceability can be defined as the link describing a relationship or dependency between two artifacts developed during the various phases of software engineering [Anq08]. It is both forward and backwards directed. A forward directed traceability refers to the post traceability, describing the deployment and use of a concept, while backwards traceability refers to pre traceability, describing the origin and evolution of a concept. [ABFG00].

Using traceability can improve the comprehension of a product line infrastructure and its product-members' development, and provides support for their evolution and maintenance [Anq08]. It also facilitates the communication between various stakeholders [HeTr03]. In conclusion, traceability plays a key role in SPLE and it allows checking the satisfaction of different requirements, which helps achieving quality [Fin94].

3 Variability mapping tools

This section presents the different variability mapping tools that we will analyse in the further sections. We will present the approaches for which we have found sufficient documentation. Also, our selection of variability mapping tools does not contain non-free tools such as Gears from BigLever [Big11], excepted if a free evaluation version exists, such as for Pure::Variants [Pur11] . The presentation of each tool will introduce how the tool define the mapping, with which type of models it can work, what are the different functionalities of the tool,etc.

3.1 FLOCOSGPL

In [GNFL09] , Gadelha et al. have defined an approach that incorporates Software Product Line techniques to allow the derivation of customized groupware, i.e. software that facilitates collaboration in groups. They used as example a groupware product line of learning object repositories named FLOCOSGPL, that stands for "Functional Learning Object Collaborative System Groupware Product Line ». In this approach, they give to the user the ability to map features of the groupware product line to elements of UML class diagrams that represents implemented classes of the product line, with a positive variability approach.

The FLOCOSGPL approach is based on the 3C Collaboration model (Figure 6) that allows identifying collaboration needs and guiding the user to select appropriate features according to their collaboration purpose [GNFL09]. It states that Collaboration depends on the interplay of three dimensions: Communication, Coordination and Cooperation. That model helped the authors to analyze the domain of their groupware product line.

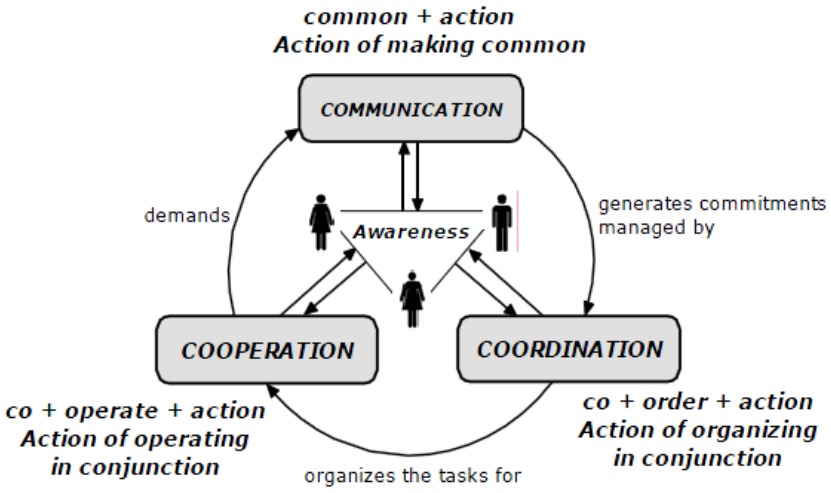


Figure 6 :The 3C collaboration model [GNFL09]

Based on this model, they defined the features of the GPL. To do so, they created a GPL-specific extended version of a standard feature model that they called "3C feature model" (Figure 7). It is a feature model that can also make the difference between

"Coordination features", "Communication features" and "Cooperation features", allowing the user to specify which part of the domain (Communication, Coordination or Cooperation) the feature belongs to.

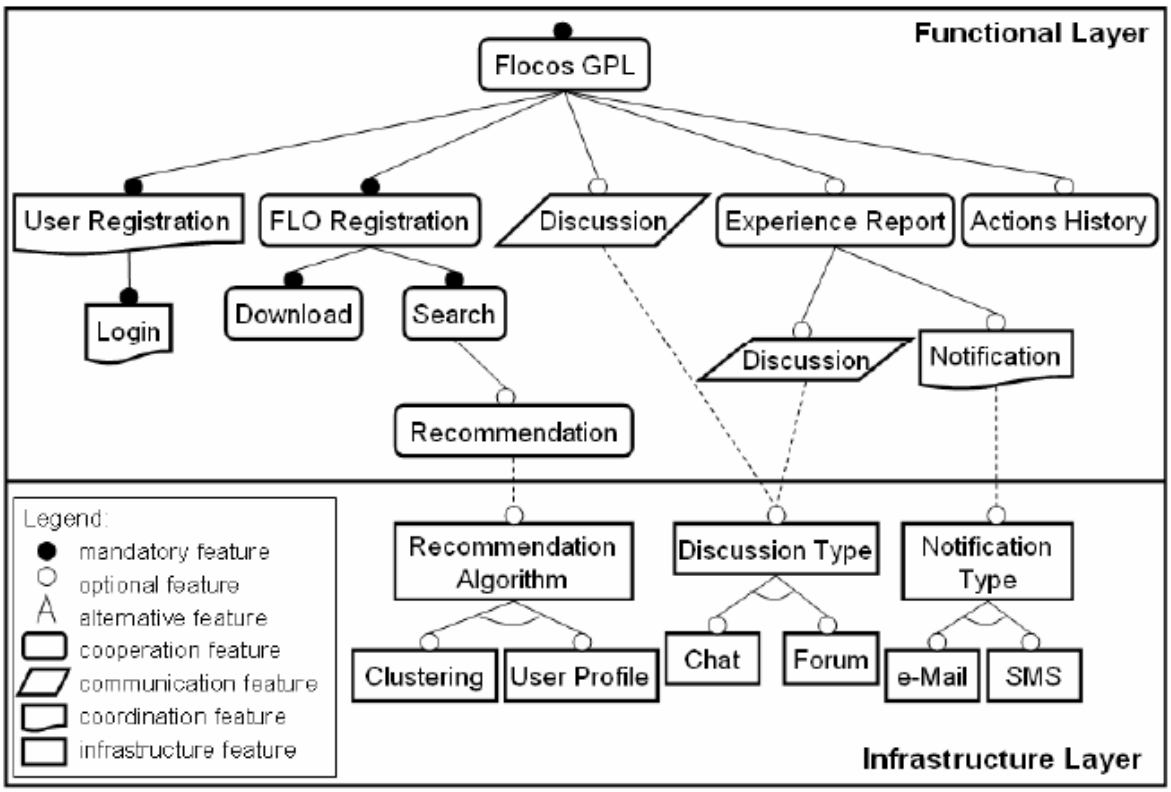


Figure 7 : 3C feature model example [GNFL09]

Architecture modeling is done using typical UML class diagrams that represents classes, configuration files and infrastructure services. Those diagrams are extended with the stereotypes "Kernel", "optional" and "alternative" in order to provide explicit variability information.

Finally, the approach requires to specify the links between features and model elements by creating a table that lists all the different features, and, for each feature, relates the elements that are linked to them. An example of a mapping expressed in the FLOCOSGPL approach is shown in Table 1. This is thus a positive variability technique, because for each selected features, we have to add the corresponding architectural elements. However, the FLOCOSGPL approach does neither provide an automatic way of generating a derived product, nor a derived model of the product. In fact, the user has to do it himself with thanks to the list. Also, there is no "core assets model" that could be used as an input of the derivation process and could be completed with specific assets depending on a feature selection. In this approach we have to add core assets elements as well.

Feature	Classes
User registration	`UserAction; UserService; UserServiceImpl; UserDAO; UserDAOMySQL`
FLO registration	`FLOAction; FLOService; FLOServiceImpl; FLODAO; FLODAOMySQL`
Discussion	`DiscussionAction; DiscussionService; DiscussionServiceImpl`
Experience Reports	`ExpReportAction; ExpReportService; ExpReportServiceImpl; ExpReportDAO; ExpReportDAOMySQL`
…	…

Table 1: mapping between features and diagram classes using the FLOCOSGPL approach [GNFL09**]**

An overview of the complete process proposed by the approach is shown in (Figure 8). It shows that the class diagrams represent the classes, configuration files and infrastructure services, which are linked to the 3C feature model thanks to a table. Those links are then used when deriving final products to create the design models corresponding to a selection of features (i.e. a variant). Those are, in the case of the FLOCOSGPL approach, customized groupwares.

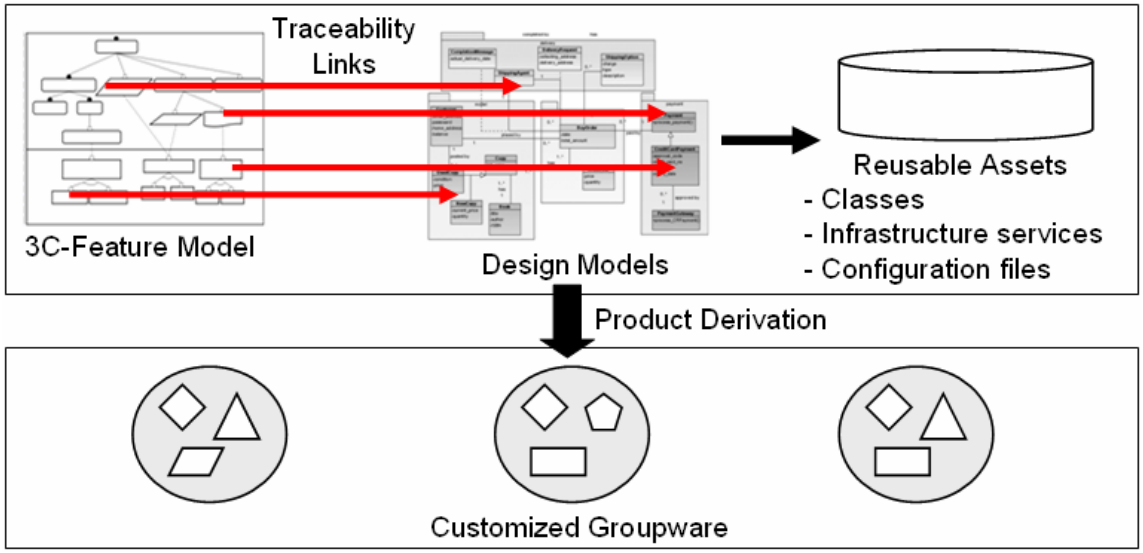


Figure 8: Overview of the FLOCOS GPL approach [GNFL09**]**

3.2 AHEAD

AHEAD(Algebraic Hierarchical Equations for Application Design) is an approach to the variability mapping problem that uses a declarative technique based on positive variability. It is based on algebraic equations.

AHEAD has not been developed especially for the variability mapping problem. In fact, it is a generic approach for adding elements incrementally, which can be java source files as well as XML parts, and thus classes of a class diagram. In [AHE11], it is described as "an architectural for feature oriented programming (FOP) and a basis for large-scale compositional programming".

In AHEAD, an application composed of different features is represented as an equation. A feature is represented as an algebraic operation, and the program, a composition of those features, is represented as a composition of those operations, thus as an algebraic equation. Furthermore, the entire family of products of a product line is an enumeration of different equations. For example, [BSR04] define a family of three applications:

app1 = i(f) // app1 has features i and f
app2 = j(g) // app2 has features j and g
app3 = i(j(f)) // app3 has features i, j, f

Where i(f) is the composition of features i and f.

The composition function is what brings transformation capabilities to AHEAD. It has to be implemented for the type of element we want to use.

Therefore, in the variability mapping problem, AHEAD could be used in order to generate class diagram in a positive variability manner. First we would have to implement the composition function for UML class diagram, or find a tool that implements it. Then we would have to create the base architecture model that would be further refined. Then, we would create the refinement classes or packages, etc. that we would like to add for specific products. Those elements would have to be mapped to a feature, which depends on the tool that we have chosen or our implementation. Finally we need to transform our base model by adding refinements that are "composed" with the base model.

The AHEAD Tool Suite (ATS) is an implementation of this approach, which is composed of different tools implemented in Java. One of these tools XAK, and its successor XAK2, are used in order to compose documents written in XML format. Those tools make use of a XAK document written in XML to operate the composition and generate a XAK document that contains the composed XML documents. The composition is thus written in XML, with operations defined in XAK such as [XAK11]:

- **<xr:at select="${XPath}"> :** This operator selects the first occurrence in the XML document defined by the XPATH request.
- **<xr:append>** : This operator appends some content at the end of the content of the selected element.
- **<xr:prepend>** : This operator prepends some content at the beginning of the content of the selected element.
- **<xr:override select="${RelativeXPath}">** : This operation overrides the content of the selected element.

With this tool, it is possible to compose XML documents. We can add XML content, or also override XML content, and thus we can use positive variability and element modification. Since UML diagrams are written in XML, it is also possible to compose UML diagrams with AHEAD. For example, in [TBD07], Trujillo used XAK to compose derived state charts for customised web portals in a model driven development approach.

3.3 MODPLFeaturePlugin

MODPLFeaturePlugin [BuDo092] is a tool that allows mapping features from a general feature model to a model of the system domain. It is a part of the Fujaba Tool Suite, which is a suite that provides support for model based software engineering. This suite can be integrated to Eclipse via the Eclipse plugin "Fujaba4Eclipse". It allows the user to create UML models, to generate java code based on a formal definition of a system's architecture and behaviour, to use model to model transformations, etc. [FUJ11].

The Fujaba Tool Suite provides support for creating and editing UML diagrams, which are defined with a specific syntax. Similarly, the mapping plugin, MODPLFeaturePlugin, requires using diagrams that have been realized in Fujaba. Therefore, this approach does not support models designed with other tools.

In order to represent the mapping, the MODPLFeaturePlugin tool makes use of annotations that are attached to the domain elements and that specify the feature's unique identifier via an instance of UMLTag class . The syntax of these annotations is quite similar to java annotations: an annotation starts the symbol "@" followed by a string literal which specifies the name of the tag (for example: "feature") and key-value pairs surrounded by parentheses (for example: "(id= FEATURENAME)") [BuDo092]. Examples of such tags can be found in Figure 9, where for example the tag @feature(id="directedDeltas") is linked to the class "ForwardDeltaStorage". MODPLFeaturePlugin thus belongs to the category of declarative techniques that make use of negative variability (because it uses a model that contains all the elements of the product line) and that make direct annotations on the model.

With this technique, model elements can possess multiple annotations and thus can express AND expressions this way. In fact, a model element will not be removed from the base model every feature specified by a tag attached to this element has been selected. Other feature expressions cannot be expressed. In fact, the AND expression is created by tagging multiple constraints, and this simple mechanism cannot be further defined to express other feature expressions. It is possible to annotate every element of the FUJABA model, such as a class, an attribute or a package. This, according to [BuDo092], "keeps the multivariant architecture manageable, but it is up to the modeler's discipline to use the feature annotations carefully".

In order to have a good usability, the tool provides some help to the user. For example, when specifying the mapping, the tool provides a way of selecting the features from a list of features that are contained in an existing feature model. It helps avoiding spelling errors and annotating elements with features that don't exist. MODPLFeaturePlugin also offers a

visualization technique that highlights in green the elements of a model that mapped to the feature that the user is clicking on. Additionally, it possesses a model checker that check constraints. For example, in Figure 9, the elements highlighted in red (or in grey in printed versions), which are the class "CVSDeltaStorage" and the tags "@(feature(id=backwardDelta)" and "@(feature(id=CVSLikeDelta)", don't satisfy a constraint, which is that the two features mentioned are defined as alternatives". This technique is also able to handle propagation rules, and thus can highlight in blue the elements that are also affected when we add a tag to another element. Finally, features that are not used in the mapping are noted "unused feature" in the feature model.

The tool is integrated into the FUJABA tool suite, which allows it to be able to derive product variant from a configuration created with "FeaturePlugin", and via the code generator tool of the suite. Figure 9 shows a part of a class diagram modelled in the Fujaba tool suite and that contains annotations to realize the mapping. Here, the tag attached to ForwardDelta is highlighted in green because the user has clicked on it in the feature tree view (not visible here) and the tags attached to "CVSDeltaStorage" are highlighted in red because they violate a constraint, which is that they are mutually exclusive but are tagged to the same element. That element will thus never appear in a derived model, because it is not possible to select both features.

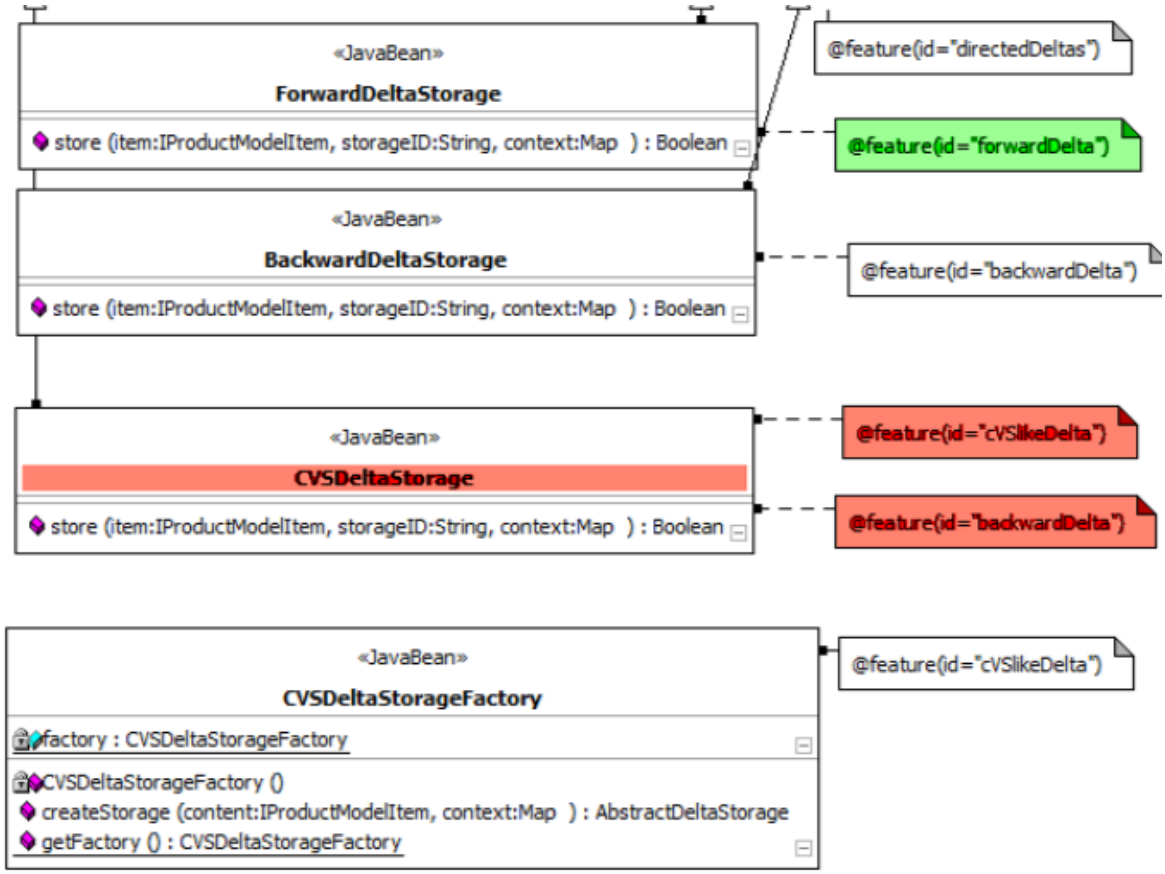


Figure 9 Example of variability mapping with FUJABA [BuDo092**]**

3.4 CZARNECKI

Czarnecki et al. [CzAn05] proposed an approach for the variability mapping problem that he called "a template approach based on superimposed variants". This approach makes a 1:1 mapping of a feature model described in fmp, a feature modelling plugin for Eclipse, to a class model expressed in the Meta-Object Facility (MOF) where direct annotations are made in order to remove the undesired elements via a transformation depending on the selection of features. It is thus a declarative approach that uses negative variability with direct annotations on the architecture model. In the paper, Czarcnecki shows how the approach can be applied to UML 2.0 activity and class models and describe a prototype implementation. According to him, "this approach is particularly desirable at the requirements level, as it directly shows the impact of selecting a given feature on the resulting model."

This method thus uses two models, a feature model and a model template. The model template is the model that we want to transform and that contains all the possible elements. This model template is annotated with two sorts of annotations, which are called presence conditions and meta expressions. Presence conditions make the link to the features in the feature model and are thus used to specify whether the element should be removed or not during the transformation. Presence conditions can be written using XPath in order to express more complex conditions that can be based on the attributes values for example. Meta expressions allow the user to compute values of attributes of an element, and are also written in XPath.

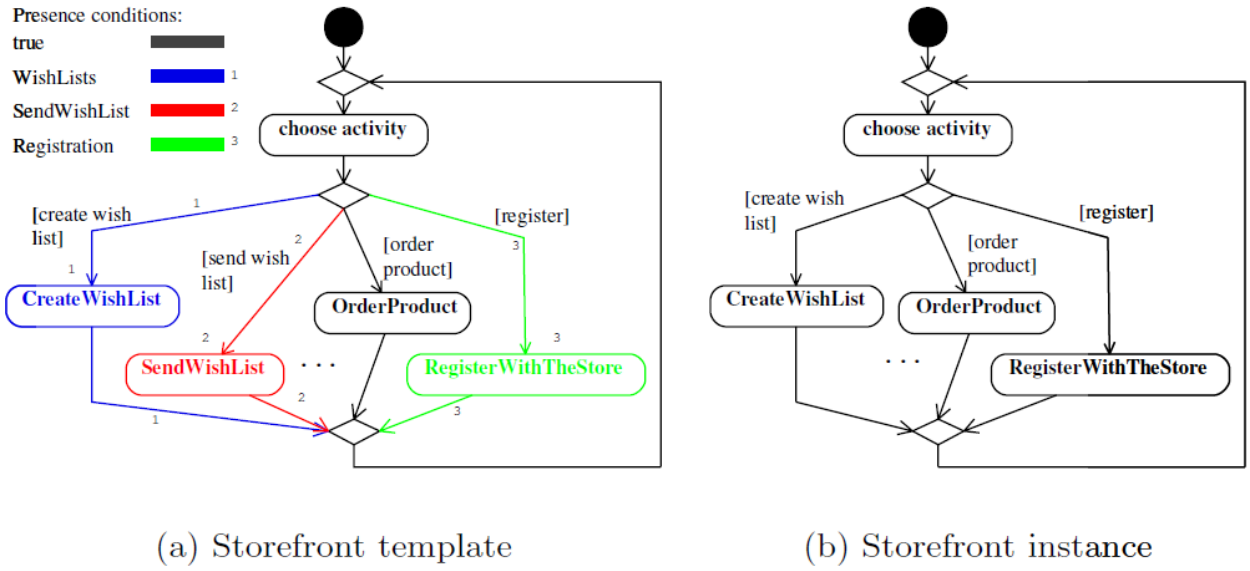
Also, this technique uses implicit presence conditions. They are defined in a separate file and are used when no explicit presence condition is attached to an element. Then, the implicit presence condition that is specified for the type of the element (each type has one) is used during the transformation to see if it has to be removed or not.

This technique also requires to define patches, which are composed of transformations that have to be applied when specific elements are removed. For example, we have to remove associations of a class when this class has been removed.

The transformation process begins with computing the meta-expressions. Then it removes the elements which have explicit or implicit presence conditions that are not satisfied. Then, patches are applied and finally the simplification is applied to the model, which removes the redundant model elements if any exists.

The prototype implemented by Czarnecki, *fmp2rsm*, is an Eclipse plug-in which integrates the Feature Modeling Plug-In (fmp) for the feature models and Rational Software Modeler (RSM), a UML modeling tool from IBM, for the architectural models. It can thus handle all UML models, but the patches and simplification instructions have only been specified for activity diagrams. Fmp allows the user to create and edit feature models as well as create a configuration model, which is basically a selection of features. It also supports the use of constraints on the features and feature attributes via XPath or propositional formulas [Cza05]. Fmp also contains a model checker that is used to verify that the constraints are not violated. *Fmp2rsm* brings an "automatic verifier" that makes sure that no ill-structured template instance can be created for a valid feature configuration [Cza05].

Figure 10 shows an example of a model template and an output model derived from this template. In this example, the presence condition of the element "SendWishList" is not satisfied by the current configuration, so it is removed from the base model during the derivation.



(a) Storefront template (b) Storefront instance

Figure 10 : Variability Mapping example (a) and output model example (b) from the Czarnecki model templates approach [CzAn05**]**

3.5 Ziadi's approach

Ziadi et al. [ZiJe06] introduce a model transformation process for product derivation using direct annotations on a UML class diagram. The transformation is here based on an algorithm that has the architecture model and a "decision model" as inputs.

The architecture model is a generic UML class diagram that will be refined by the approach. In fact, the approach requires specifying the elements that are not mandatory, and are thus optional and don't belong to the core assets, by an "optional" stereotype added to the class, package, attribute or operation. It also requires specifying "variation" and "variant" stereotypes where variation points occur. The idea is to define variation points as abstract classes and the different variants as concrete classes. In order to derive a product from such an architectural model, we have to specify which concrete class will implement each variation point (it can be zero or multiple classes as well). Figure 11 shows an example of a refined architectural model.

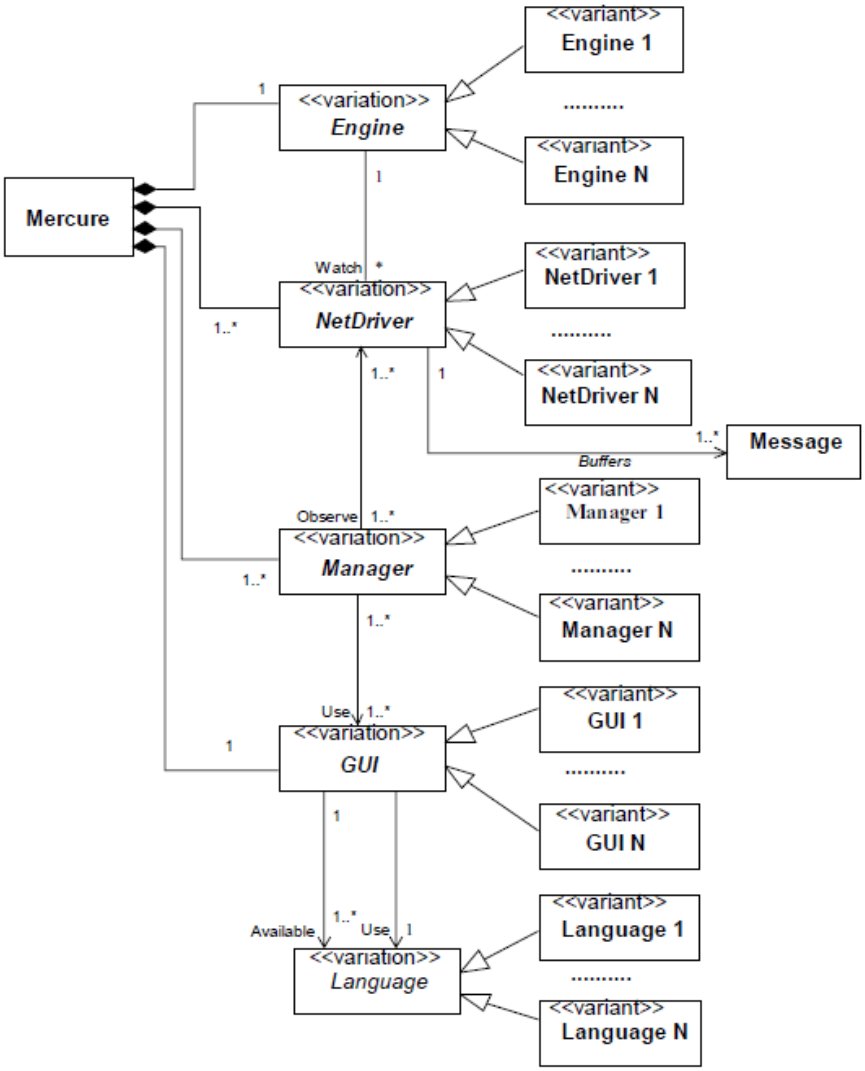


Figure 11: Refinement of a UML class diagram by adding stereotypes [ZiJe061**]**

This approach also needs to specify the constraints and relations between features (such as requires) in the OCL language. For example, the constraint that specifies that one feature (for example "NetDriver1") implies the presence of another feature (for example "Engine1") is written as follows [ZiJe061]:

context model inv:

self.presenceClass('NetDriver1') **`implies`** self.presenceClass('Engine1')

After having modelled the features and the architecture, we have to specify the selection of features specifying products, in a model called decision model. Ziadi's approach uses factory models in order to do that. Thus, it doesn't require a feature diagram, but directly map the elements in the factory thanks to the stereotypes. For example, for a product that would contain the GUI "GUI1", the factory method of the concrete factory created for the product will be stereotyped with "GUI1". Figure 12 shows the abstract factory Mercure_Factory [ZiJe061] and three concrete factories that correspond to concrete products of the SPL.

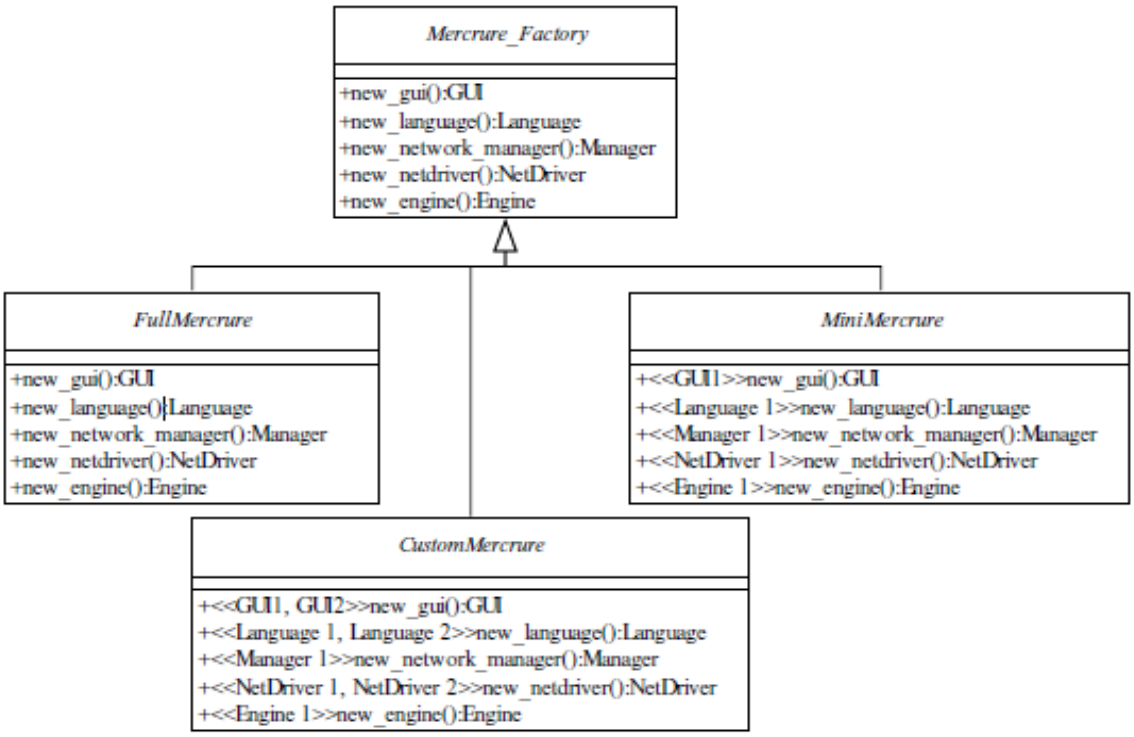


Figure 12: The abstract factory Mercure_Factory and three concrete factories that corresponds to concrete products of the SPL [ZiJe061]

Then, we can derive the product models thanks to the derivation algorithm, which generates the product class diagram of the architecture. It is based on three steps [ZiJe061]:

- *Step 1: Variant classes selection*. The first step consists of selecting variant classes. It creates a list of the classes that are selected.
- *Step 2: Model specialization.* Here we remove from the model the classes that are not selected
- *Step 3: Model optimization.* It consists of Inheritance optimization, which is applied when there is only one concrete class inheriting from an abstract one. In this case the abstract class is omitted and replaced by the concrete one.

Figure *13* shows ZIADI's algorithm for model transformation in pseudo-code, which defines the three steps of the transformation.

```
algorithm: DeriveProductModels()
Input :  PL_classDiagram: Model, aConcreteFactory: Class
Output : Product_classDiagram: Model
 – Step 1: Variant classes selection
selectedVariantsList:=∅;
for each factory method in aConcreteFactory do
  initiate definedVariantsList to
  significant stereotypes of the factory;
  if definiedVariantsList is empty then
    selectedVariantsList.add(all sub-classes of the returned type of the
    factory);
  else
    selectedVariantsList.add(definedVariantsList);
  end if
end for
 –Step 2: Model specialization
for each variant class C in PL_classDiagram do
  if (the class name of C not in selectedVariantsList) and (names of all
  sub classes of C not in selectedVariantsList) then
    delete the class C from the PL_classDiagram;
  end if
end for
 –Step 3: Model optimization
delete unused concrete factories;
optimize inheritance;
Product_classDiagram:= PL_classDiagram;
return Product_classDiagram;
```

***Figure* 13: ZIADI's algorithm for model transformation in pseudo-code [**ZiJe06**]**

3.6 FEATUREMAPPER

FeatureMapper [Fea11] is a tool in development at the Software Technology Group of Technische Universität Dresden, Germany. It allows defining mappings from features to model elements of a solution domain model by using "mapping models". It has been developed as an Eclipse plugin.

We can use FeatureMapper for the variability mapping problem in two ways: we can use FeatureMapper only, or we can use it in combination with Pure::Variants.

Pure::Variants is a plug-in for the OpenSource Eclipse Integrated Development Environment (IDE) which allows the user to support each phase of the software product-line development process (except from the test processes, but there is a work in this direction) [Pur] . It allows representing the problem domain of the product line using feature models, which contains the commonalities and the variabilities of the domain. This is done using the Feature Model Editor of Pure::Variants (Figure 14). The feature models are expressed in Pure::Variants feature model language.

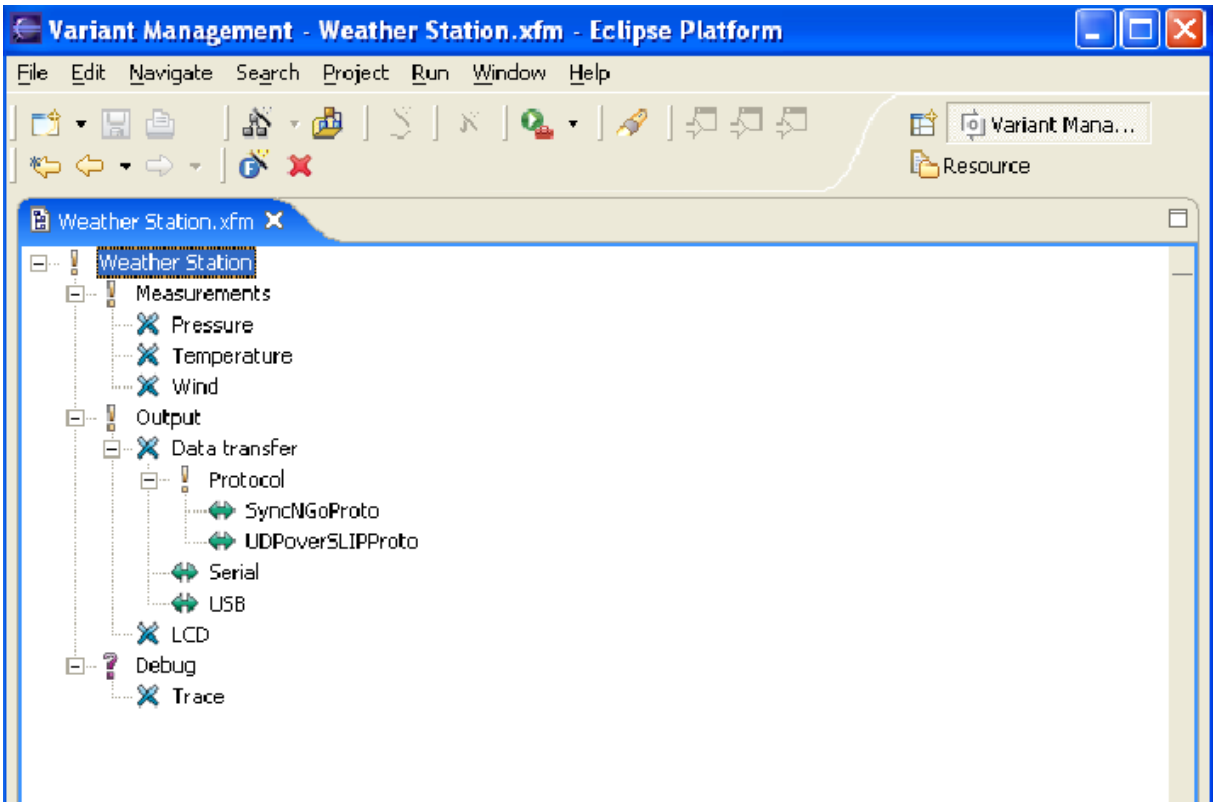


Figure 14: Pure::Variants Feature Model Editor

Pure::Variants also allows representing the solution domain via a "family model". They define architectural elements, such as configuration files, java classes and documents. Then, a mapping can be created form the feature model to the family model in order to describe how the products in the product line will be assembled or generated from pre-specified components. Also, this tool allows to represent a single problem from the problem domain (i.e. a single product from the product line) using Variant Description Models, that are basically an extension of feature models, where the user can select the features that he wants to add in the product. The tool provides here a model checker that checks if the constraints of the feature model (such as a mandatory feature or mutually exclusive features) are respected. This is thus another way of specifying the mapping between features and solution domain elements, but here the solution domain elements are not expressed in a UML model, but in a "family model". Therefore, this mapping technique does not belong to the scope of our analysis. But, if we use Pure::Variants in combination with FeatureMapper, we can create mappings from the Pure::Variants feature models to the EMF based solution models used by FeatureMapper with a FeatureMapper mapping model, and also benefit from the variant description models.

For both methods, FeatureMapper is able to handle as solution domain model all models that are expressed in EMF language, including UML2 [Hei09]. It is also able to map several models in one mapping. In fact, FeatureMapper is based on the Eclipse Modelling Framework (EMF) that provides the Ecore metamodelling language which is used to specify the abstract syntax for arbitrary modelling languages [Bal08]. Thus, the modelling of the

solution space is not bound to any concrete language and existing EMF-based modelling tools (e.g. TOPCASED) can easily be integrated.

In the mapping model, we are able to create rules that link one feature (or one feature expression) with model elements. A rule thus consists of one or more features from the feature model, and the list of all models elements that are linked to the rule. For example, Figure 15 contains the rule "CallCenterEmployee", which maps the feature "CallCenterEmployee" to some model elements of a UML class diagram. Those rules are used during a transformation to remove the elements that are mapped to features that are not selected in the variant description used for the transformation. It thus belongs to the negative variability techniques that use a separate annotation model. Both methods provide their own transformation functionality.

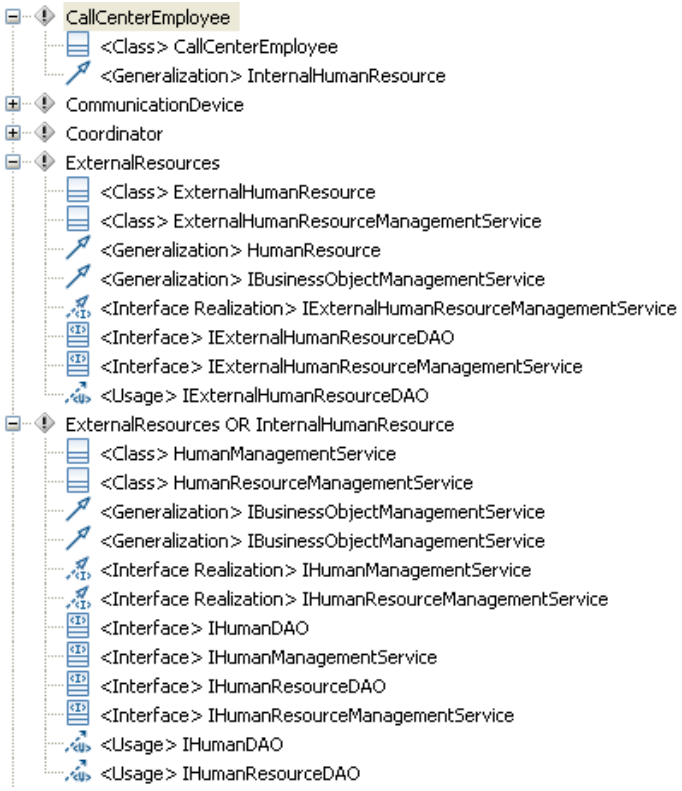


Figure 15:Example of mapping model created with Pure::Variants and featureMapper

The difference between both methods is that we are able to reuse feature models and product configurations (as known as "variant description models")of Pure::Variants if we use it along with FeatureMapper. We can thus create mappings between existing feature models and solution domain model elements and use these mappings on a product configuration. Otherwise, if we use FeatureMapper alone, we have to create the feature model and the product configuration with FeatureMapper. An example of such feature model is given with Figure 16. In order to specify a selection of feature, we have to create a mapping that contains all the features, copying it and then removing the mapping of the features we don't want to select from the copy.

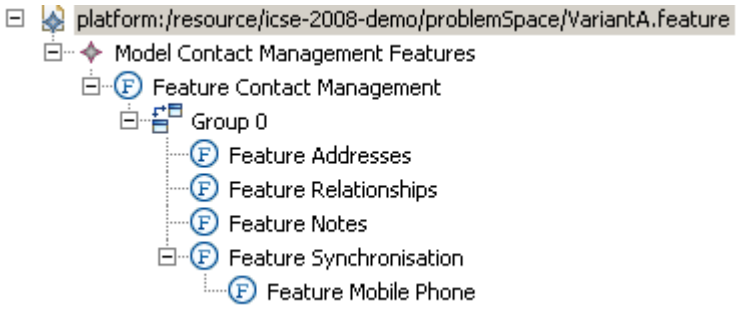


Figure 16:Feature diagram created with FeatureMapper

Finally, the tool FeatureMapper provides three visualization techniques, that are also usable with Pure::Variants: depending on a variant description, we can see which elements of a model are mapped to the features selected in the variant, we can also see which elements are mapped in a rule and which are not (which means that they are considered as "core assets" and will be present in the derived model of all variants), and we can also map colours to rules and benefit from a coloured model, where elements are coloured with the colour that is mapped to the rule where the element as also been mapped. FeatureMapper also provides a model checking feature that checks if the referenced features and model elements exist, otherwise it informs the user that the mapping model is invalid.

3.7 XSLT script with Pure::Variants

The tool Pure::Variants offers several mapping techniques. It can be used with FeatureMapper to propose a declarative variability mapping techniques that uses negative variability with a separate model, as we saw earlier. It can also use a generic model transformation technique: XSLT.

XSLT is a language for transforming XML documents into other XML documents, or text documents, or HTML documents, etc. .It is thus able to transform or generate models, such as an architectural elements class diagram, depending on another model such as a feature model. It is usable with Pure::Variants thanks to Pure::Variants extension functions for XSLT that, among others, allows to get information about elements and to know if a feature has been selected or not. This language then allows us to parse a model in order to analyse each element of the model.

The XSLT language has a "formatting vocabulary" [XSL11] that can be extended, such as Pure::Variants did. With XSLT, a transformation script such as an operational variability mapping is thus represented in a XML file. In this file, we first write some headers to specify some options of the transformation script. For example, if we want the output to be an XML file with indentation, we write the following part of XML code:

```
<xsl:output method="xml" indent="yes"/>
```

Then we specify the type of the input model. The specification of which model will be taken as input is made in the transformation configuration interface of Pure::Variants. With

this approach, we can take every model as a solution domain model as far as it is defined in XML.

Then we can write the XML generation code. It consists of writing the output XML code in the transformation script. We can write all the code ourselves, but we can also use XSLT functions to make it more useful. In fact, XSLT used with Pure::Variants allows us to parse the input model and to take decisions (what to write,…) depending on the elements of the input model. We can for example directly write conditions on the existence of a feature in the feature model specified. It is also possible to get the type, value of an attribute,.. of a Pure ::Variants model element, and take decisions depending on these types or values.

XSLT is thus useful to generate XML documents or to transform them (for example, we can copy only the elements of a XML documents that we want, which is then a negative variability approach if used for a variability mapping). The generation of a UML diagram from scratch is way more complex though, as we have to write all the low-levels details of the XML file that represents the diagram. The same can be said about adding new model elements in a diagram, and therefore a positive variability approach with XSLT is very complex for UML models.

3.8 CVL

The Common variability Language, CVL, is a language issued by the OMG [OMG11]. It allows expressing the variability of a product line and transforming models to obtain product specific models. It is based on Aspect oriented modelling (AOM). It can be used with the CVL tool support that consists of Eclipse plugins.

CVL uses three models. First, the base model is the model that we want to transform. It can be of any type of model, from UML models to specific models described in a DSL language (A DSL is a programming or modelling language dedicated to a particular problem domain [Sven10]). For example, in [Sven10], they apply CVL on TCL, a language for modelling train stations. Second, the variability model defines the variability on the base model. In fact, it describes all possible variations using the CVL variability mechanisms. Third, the resolution model is used to make a selection of the variations that we want to include in the product model. The base model can have several variability models, and a variability model can also have several resolution models. Finally, a “CVL model” is the composition of a variability model with its resolution models. Figure 17 shows an example of a CVL model viewed in a CVL model editor.

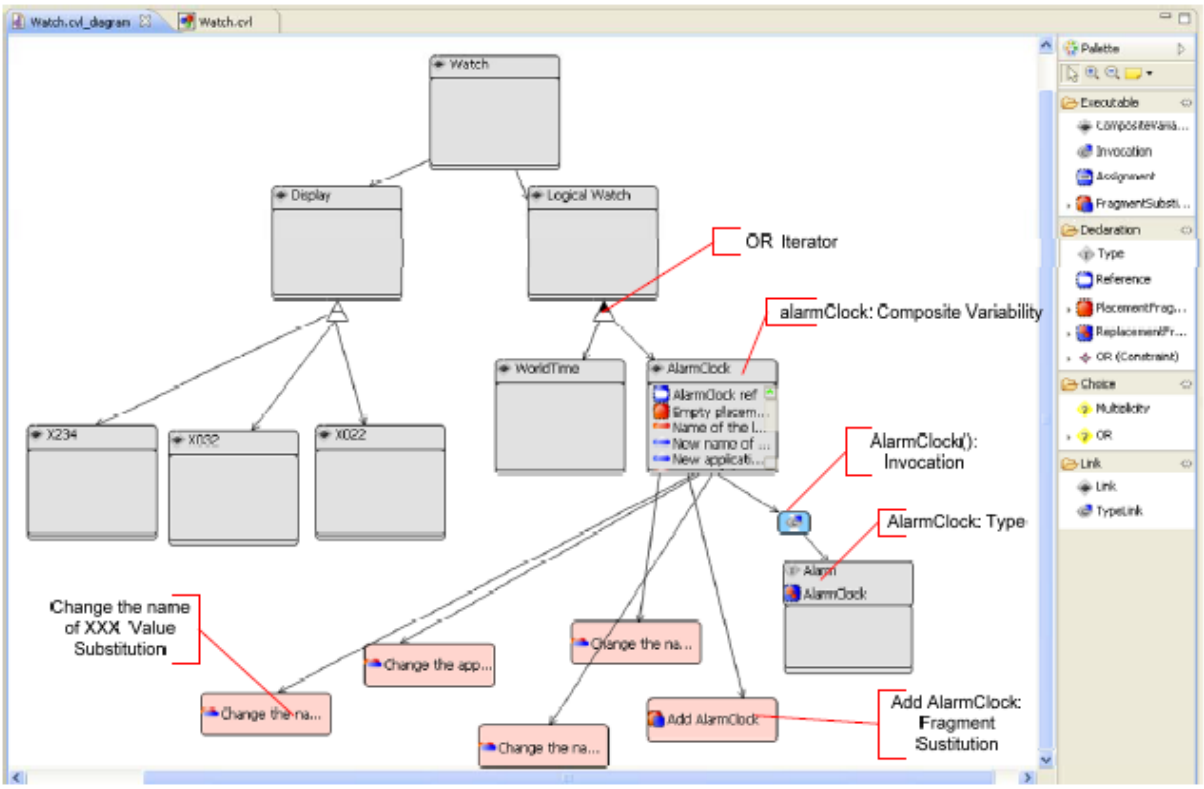


Figure 17: CVL model example [CVL**]**

CVL uses two layers: the feature specification layer and the product realisation layer. The feature specification layer expresses the features of the product line in a similar way than a feature diagram. It uses concepts like type, composite, variability, constraint and iterator, that can express the same than ordinary feature diagrams concepts such as mandatory/optional, OR/XOR decompositions, cardinalities, relations, etc. . The product realisation layer is used to define the mapping between the feature specification and model elements via transformation operations, based on the three operators, "value substitution", "reference substitution" and "fragment substitution".

CVL is a generic language that allows mapping features from to variability model to elements of the base model (which can be of any type) with the use of resolution models . The different operations available for the mapping are provided by the three generic operators: value substitution, which consists of changing the value of a literal, reference substitution, which consists of changing the value of a reference, and fragment substitution, which consist of replacing a fragment of the model by another one. A fragment of a model is a group of objects from this model, and it thus ranges "from a link or an object to a group of object of any complexity" [Sven10].

CVL thus enables the user to customize model elements, remove, replace or add new elements in a diagram, which means that it is able to use negative variability techniques as well as positive variability techniques.

The CVL tool includes a graphical editor that allows modifying the CVL model (the variability model and the resolution model). It also has a resolution model generator, which generates a resolution model based on a variability model [CVL]. The generated model depends on the elements from the variability model that the user has selected. The CVL tool also allows the user to transform automatically a model to a product-specific mode. The tool editors enable the user to select and highlight the elements so that a user can use the editors without knowing the details of CVL. It can also support APIs that support integration between base language editors and the CVL editors. This integration allows the user to highlight how base model elements are affected by a substitution [Sven10].

3.9 VML*

VML* is an operational approach to the variability mapping problem that allows the user to specify its own customized language depending on the models he wants to analyse and generate. It is available as an Eclipse plugin: the VML tool suite.

In order to create a mapping between a feature model and a solution domain model, VML* uses a "mapping language". For example, [Hei09] uses the customized languages VML4ARCH and VML4RE . VML4ARCH is a language dedicated to generate architectural models such as class diagrams, whereas VML4RE (Figure 18) is dedicated to models used in requirements engineering. Every VML language is based on a customisable infrastructure, which is adaptable for every target model. Each language consists of a set of actions that are model transformations such as remove, add, merge,… and that are executed depending on the features selected. Thus, VML* is a generic approach that is applicable to any EMF model, but there are additional costs when the language for a typical model has not already been defined. According to [Hei09], "the development of a new VML* language starts to be cost-effective when this language is applied to the development of several SPLs or a same SPL evolves and the VML* language helps to reduce the maintenance cost".

```
01  // Define a new language called vml4req
02  vml instance vml4req {
03    // This section defines the type of variability model and how to access it
04    features {
05      metamodel "/bin/fmp.ecore"
06      // Extracts all variability units from a variability model
07      function "getAllFeatures"
08    }
09
10    // This section defines the type of target model and how to access it
11    target model {
12      metamodel "UML2"
13      type "uml::Package" // Metamodel type of a model
14      // Function to interpret pointcut designators
15      function "dereferenceElement"
16    }
17
18    // Importing plugins and external specifications
19    ...
20
21    // Syntactical definition of available actions
22    actions:
23      createInclude {
24        params "List[uml::UseCase]" "List[uml::UseCase]"
25      }
26      insertUseCase {
27        params "String" "uml::Package"
28      }
29      ...
30
31    // Definition of available evaluation aspects
32    aspects:
33      transformation { // Evaluation for product derivation
34        // Defines adapter for product-configuration access
35        features {
36          type "String"
37          function "getAllSelectedFeatures"
38        }
39        // Definition of the semantics of actions as model transformations
40        createInclude {
41          function "createIncludes"
42        }
43        insertUseCase {
44          function "createUseCase"
45        }
46        ...
47      }
48      tracing {
49        createOps "create* (*)"
50        removeOps "remove* (*)"
51      }
52  }
```

Figure 18: VML4RE specification [Hei09**]**

In VML*, the mapping is represented in a text format that depends on the customized language chosen. The possible transformations actions are in fact defined in the specification of the language (called language instance descriptor). The mapping text makes use of those actions in the rules that it can define which are called “variants” in VML. For example, in Figure 19, the “variant” (=rule) rescue is mapped to the feature Rescue and uses actions such as “insertusecase”, which is defined in the VML4RE language specification (Figure 18). Those variants can also support feature expressions, such as seen in Figure 19 with the variant “and (Rescue,Flood)”. The language instance descriptor allows the user to specify only the

parts of the language that have to be customized, while the other parts are generic parts that are reused.

```
01  import features <" CrisisSystemFlood.fmp">;
02  import core <"/CarCrisisSystem.uml">;
03
04  concern CrisisSystem {
05
06  variant rescue for Rescue {
07    insertUseCase ("ExecuteRescueMission", "CrisisManagementSystem");
08    createInherits ("CrisisManagementSystem::ExecuteRescueMission",
09  "CrisisManagementSystem::ExecuteMission");
10
11    trace ("ExecuteRescueMission::TransmitInjuryInformation");
12    trace ("ExecuteRescueMission::IdentifyVictim");
13    trace ("ExecuteRescueMission::RetriveMedicalRecord");
14    trace ("ExecuteRescueMission::AdministerFirstAid");
15    trace ("ExecuteRescueMission::BringToHospital");
16   }
17
18  variant public_hospital for PublicHospital {
19    insertActor ("HospitalResourceSystem", "");
20    insertActor ("FirstAidWorker", "");
21    createAssociation (or( "HospitalResourceSystem", "FirstAidWorker"),
22  "CrisisManagementSystem::ExecuteRescueMission");
23  }
24
25  variant for and ( CarCrash,Rescue) {
26    createAction ("RemoveVictimFromCar", "ExecuteRescueMission");
27    connectActivityElements("ExecuteRescueMission::dn2",
28  "ExecuteRescueMission::RemoveVictimFromCar", "[isVictimLockedInCar]");
29    connectActivityElements("ExecuteRescueMission::RemoveVictimFromCar",
30  "ExecuteRescueMission::FinalNode", "");
31    connectActivityElements("ExecuteRescueMission::dn3",
32  "ExecuteRescueMission::FinalNode", "[false]");
33  }
34
35  variant for not Rescue {
36    removeElement("ExecuteRescueMission");//removes the core activity model
37  }
38
39  variant for and (Rescue ,Flood) {
40    createAction ("RequestSpecialAssistance", "ExecuteRescueMission");
41    connectActivityElements("ExecuteRescueMission::dn2",
42  "ExecuteRescueMission::RequestSpecialAssistance", "[isVictimHandicapped]");
43    connectActivityElements("ExecuteRescueMission::RequestSpecialAssistance",
44  "ExecuteRescueMission::FinalNode", "");
45    createAction ("RecoverVictim", "ExecuteRescueMission");
46    connectActivityElements("ExecuteRescueMission::dn3",
47  "ExecuteRescueMission::RecoverVictim", "[false]");
48    connectActivityElements("ExecuteRescueMission::RecoverVictim",
49  "ExecuteRescueMission::FinalNode", "");
50  }
51
52  variant for Observe {
53    insertUseCase ("ExecuteSuperObserverMission", "CrisisManagementSystem");
54    createInherits ("CrisisManagementSystem::ExecuteSuperObserverMission",
55  "CrisisManagementSystem::ExecuteMission");
56    insertActor ("SuperObserver", "");
57    createAssociation ("SuperObserver",
58  "CrisisManagementSystem::ExecuteSuperObserverMission");
59  }
60
61  variant for not (Observe){
62    removeElement("ExecuteSuperObserverMission");
63  }
64  }
65
66  order (rescue, public_hospital );
```

Figure 19: Variability mapping using VML4RE [Hei09**]**

When we specify the mapping, the first thing to do is import the feature model and the core assets model that will be transformed. Then, we can define actions that will be executed depending on the selection of features, thanks to the variants (the rules) that we can specify. We can for example add or remove elements from the model, which means we are able to use the positive variability technique as well as the negative one.

VML* also offers the possibility of specifying trace links. They can be used to visualize the mapping between features and model elements. The AMPLE Traceability Framework (ATF) offers several visualizations to show trace links.

4 Literature-based Comparison of the tools

In the last chapter, we introduced different approaches to the variability mapping problem. Each of these approaches has different characteristics and uses different techniques. Also, this selection covers every type of variability mapping techniques that was described in section 2.7, where we presented an overview of the different variability mapping techniques.

However, our goal is not only to present them, but also to compare them and describe what are their capabilities compared to the other tools. To do so, we will present in this chapter a literature-based comparison of those tools, which will be based on comparison criteria that we will define in section 4.1 and that we will apply on the tools, based on their documentation.

4.1 Comparison criteria

In order to compare the selected variability mapping tools, we first define the different criteria that we will apply for the comparison. We regrouped the different criteria into four main criteria. They are the expressiveness of the tool, the adaptability of the tool, its usability and its maturity.

4.1.1 Expressiveness

Our first comparison criterion is called "Expressiveness". It contains questions that focus on what we can express explicitly within the different models or languages used by the tool. Some concepts can be expressed indirectly or implicitly by the tools, but here we are in fact focusing on an "explicit expressiveness", and not a mathematical expressiveness, such as defined in [SHT06]. In fact, an optional feature can also be expressed indirectly using feature cardinalities for example, such as with FeatuRSEB [GFdA98] feature diagrams. Expressing the different concepts explicitly can among others make the diagrams more readable. Therefore, when we will evaluate the expressiveness of the different tools, we will answer the following questions:

What concepts is it possible to express explicitly with the feature representation used by the tool? So, for this criterion, we will analyse if the feature representation can explicitly express the following concepts, which are listed in [IKJ10] :

- Mandatory feature
- Optional feature
- And decomposition
- Or decomposition
- Xor decomposition
- Relations between features
- Explicit marking of variation points
- Feature cardinalities

We will also analyse if the tool is able to express the selection of some features in order to specify variants, which is often done with a model called "variant model".

What are the domain model languages supported? Is the tool able to take a general UML model as input? This question is thus about the "genericity" [MKR06] of the tool. In

[Hei09], the authors argue that an approach that can handle more domain model languages is more usable.

What are the different concepts that we can express in the mapping between features and domain model elements? We will therefore use questions such as: What are the different variability mechanisms (negative variability, positive variability and element modifications) that the tool offers? Is it possible to use feature expressions (i.e. boolean expressions over feature names") in the mapping?? Is it possible to reuse rules in other rules (thus making use of "reuse mechanisms" [CzHe03])? What kind of model elements is it possible to map?

4.1.2 Adaptability

Our second criterion is called "Adaptability". In this criterion, we regrouped to questions:

Is the tool usable on a "real life SPL" [Hei09] which makes use of very large feature models and domain models? Thus, is it "scalable"? We will evaluate this scalability with qualitative arguments, as it is hardly feasible to apply a real life example on every tool studied here[4]. An example of such argument is the type of variability mechanism used. In fact, if the tool only supports negative variability, it requires the solution domain model to contain every possible model element that one product of the product line could possess. Therefore, this domain solution model will big very large, which will cause usability problems. If the tool also supports positive variability, the solution domain model will only be composed of the mandatory elements, and will be therefore of a much smaller size.

Does the tool bring support for model evolution? For example, if the mapping has to be changed, does the tool allow us to implement the different changes without too much difficulty? We will also evaluate the case of the evolution of the feature model and of the domain model.

4.1.3 Usability

Our third criterion is called "Usability". In this criterion we gathered questions about the functionalities of a tool that make it easier to use. Those questions are:

Is it able to provide automation to the derivation process? This question, also discussed in [Hei09], is based on the idea that a key feature of a variability mapping tool is to derive product models. Therefore, if the tool doesn't bring any automation to this, and thus if the user has to process the derivation himself, the tool's scope is too limited.

How difficult is it to create the mapping? In this section, we will answer this question with qualitative arguments based on the tools documentation (in section 5, we will present a more quantitative approach to the problem). Those arguments include the accessibility of the mapping model [Hei09], which kind of checking function and other support such as process guidance and user assistance (for example feature highlighting, code completion) is offered by the tool [PfPi08], and also if the tool is able to generate traceability links. In fact, according to [CzHe03], it is useful "in performing impact analysis, synchronization between models, model-based debugging and determining the target of a transformation".

[4] However, in section 5, we will apply a test case of a medium size on the tools.

4.1.4 Maturity

Our fourth criterion is called “Maturity”. This criterion evaluates the level of maturity of the tools. The tools can only have been suggested by researchers, applied on examples, or they can have been applied on industrial cases. They can also be in development, or their development can be stopped, or they can be in a stable status but still evolving. Mature tools are safer than not mature ones, because their development status is stable, and they are less prone to bugs. Also, mature tools are often more documented. We will evaluate this criterion based on the documentation found for each tool.

4.2 Comparison of the tools

Now that we have defined the different criteria to use, we will apply them on each tool that we presented in chapter 3 in order to compare them. At the end of this chapter, we will also make a resume of the comparison and draw first conclusions about the different approaches.

4.2.1 FLOCOSGPL

Expressiveness

In the FLOCOSGPL approach, the features are represented via a "3C feature model". But since there is no tool or implementation of the mapping approach (it is just a list of features and linked elements in a table), the user could use another feature model as well. Thus, for the features, the expressiveness of the approach only depends on the feature model used. As far as the 3C feature model is concerned, it can express mandatory, optional and alternative features. The AND decomposition cannot be directly expressed but we can use a group of mandatory children to do so. The OR decomposition cannot be expressed and feature cardinalities cannot be expressed too. Also, it cannot express relations between features, such as "requires" and "excludes", and it is not possible to mark variation points explicitly. Finally, this "3C feature model" is able to distinguish "coordination features", "communication features", "cooperation features" and "infrastructure features".

Unlike some other methods, there is no model to represent a selection of feature. This selection is made "by the hand". This is to be expected since the FLOCOS approach does not have automatic derivation feature, which could benefit from such representation.

In [GNFL09], they mapped features to a UML class diagram, and they then use the table representing the mapping to select classes in a Spring configuration file [Spring]. But it could also have been another type of model since the only thing required is to be able to identify the different model elements with a name. In theory, this approach thus benefits from a perfect genericity concerning the different domain models usable.

The mapping proposed by FLOCOSGPL can only use positive variability as variability mapping mechanism, meaning that the user can only add desired model elements. He thus has to start with a core assets model and add elements to it and it cannot modify elements, or remove them. The representation of the mapping doesn't allow the user to express feature expressions as well, so the user has to create mappings that concern only one feature at a time. Finally, in [GNFL09], they only map UML model classes, but it would be also possible to map other elements such as classes' attributes, if we are able to distinguish them by an identifier. We could for example prefix the attribute name with the name of the class that contains it.

Adaptability

The approach of FLOCOSGPL uses positive variability, which is better than negative variability for the scalability problem. The approach could use negative variability as well, as it only relies on a mapping table. So, instead of adding referenced elements linked to a selected feature, we could also remove elements that are referenced in the table but linked to a non selected feature

Also, the fact that FLOCOSGPL has no tool support make it less scalable, because every transformation must be made by hand, and then it would take much time to realise a transformation of a large model. Moreover, the table that the user would have to create and handle would quickly become too large and unreadable.

Because FLOCOSGPL uses a table to represent its mapping, and because there is no tool support for it, applying changes in the mapping can be hard to do, for example if the name of an element changes, we have to read the table thoroughly to spot the element in the table and modifying it.

Applying a change in the other models will result in the same problem, because if a change appears on the feature model or on the solution domain model, we have to change the table as well. Features can be spotted in the table without too much difficulty. But the solution domain model elements can be everywhere in that table and in multiple places, and thus it is not easy to remove or modify one of them (unless we use a text editor with a find function).

Usability

The FLOCOSGPL approach consists of creating a table that contains the mapping and is not supported by a tool. Consequently, it does not provide a mean of deriving products automatically. This is a big lack of usability of the approach, especially for larger product lines, where deriving products manually will need a lot of effort. For small product lines, however, the approach could be used, as creating the mapping is easy and the derivation would here need a smaller amount of effort. This lack of automation in the process is known by the authors of the approach, and they suggest tools like Pure::Variants to tackle the problem [GNFL09].

Also, without tool support, the user cannot make use of any function like spelling checks, consistency checks or existence checks, making the approach less usable than others. For example, even if the user makes use of a feature model that is able to express relations between constraints, he has to verify himself that the constraints are not violated when creating the mapping.

Maturity

The approach of FLOCOSGPL is an approach that, as it is, does not require any tool support. It has only been described once and tested on a small case study. The main concern of the authors was to implement a GPL and not to explore the mapping problem, which they decided to handle simply with a table. This approach is therefore not mature.

4.2.2 AHEAD

Expressiveness

In the AHEAD approach, the features of the product line are not expressed via a graphical feature model, but instead it uses a feature model translated into propositional logic and algebraic equations where the variables of the equation represent model parts (in our variability mapping problem) that are implicitly mapped to an abstract feature. Therefore, the

representation of a feature configuration is a specific equation, and the different final products of a product line are represented by a set of equations. With these equations, we cannot express the same concepts as with a feature model. Here, the features are just mapped to parts, and there is no relation or hierarchy between them, such as we can find in a feature model tree representation. Therefore, we can say that AHEAD lacks of a good mean of feature representation.

The approach handles the variability mapping problem by being able to compose XML parts with the XAK plugin. It is therefore able to derive product models that are expressed in any type, as long as it is written in XML. Thus, the genericity of AHEAD is very good.

In AHEAD, the main variability mapping mechanism proposed is the positive variability. Here the user has to define the core assets model and the refinement elements, as well as the composition function, before having the ability to transform the model by adding refinements. The XAK plugin allows the user to add XML content with the "append" and "prepend" operators, but also modifying XML content with the override operator.

The expressiveness of this approach also depends on the composition function that has to be implemented. However, the approach does not allow the user to directly express feature expressions. He can only add one group of refinements parts at a time. But he can create as many parts as he want, and therefore create parts that would correspond to feature expressions. Also, because it composes XML parts, it is possible to map every kind of model element.

Adaptability

AHEAD uses positive variability techniques as well as elements modifications, which is better than negative variability in the scalability point of view, because we handle a lighter base model. However, it does not use a graphical representation of the features like some other approaches, making it difficult to have an overview of the product line if it is of a larger size. We can also use a separate representation of the features such as in [TBD06], but there is no link to the tool and it is therefore not ideal. Also, creating a composition equation of a large number of features can be hard too, especially because we don't have information about feature relations in AHEAD and because the composition is about low-level XML modifications that are hidden from the user.

With AHEAD, if the mapping has to be changed, we have to implement the changes in the corresponding XAK refinement definition written in XML. Also, if we have to delete an element from the solution domain model, we have to make sure that it does not appear in the refinements anymore. The tool does not bring support for those tasks.

Usability

The AHEAD tool suite used with the XAK plugin does bring automation to the derivation of product models. But the edition of the XAK files is made via a XML editor that does not bring any support function like other approaches do. Moreover, there is no feature model supported, and there is no mean of viewing the model elements in a graphical manner. In fact, we can only see them as XML code, since XAK focuses on XML parts composition that could be something else than model elements. Also, the mapping is less easy to create

than in other approaches like FeatureMapper, because with AHEAD we have to handle the composition of the parts as well as the XML based refinements, whereas with tools like FeatureMapper the mapping is created using the graphical user interface by clicking on model elements. Therefore, the AHEAD approach lacks of usability.

Maturity

The general concern of AHEAD is to create compositions of software artefacts, while the composition of XML artifacts, which allows the user to handle the variability mapping problem, has been developed as an AHEAD plugin, XAK. AHEAD has been developed as the AHEAD tool suite, which is a mature tool, and was last updated in 2008. However, the XAK plugin has only been developed as a research project and is not published yet (on [XAK11], they say "We may release these tools open-sourcely when paper becomes published."). It has been tested for our problem in [TBD07] on state charts, but in a small scale problem created by researchers.

4.2.3 MODPLFEATUREPLUGIN

Expressiveness

The FUJABA plugin MODPLFEATUREPLUGIN supports its own type of feature model only (the one developed in the FUJABA tool suite), which is described in [BuDo091]. It is able to express mandatory features, optional features and AND, OR and XOR decomposition. However, it is not able to express feature cardinalities and constraints between features. Also, it does not provide a mean of explicitly marking variation points. The plugin does not allow the user to create different product configurations. In fact, the selection of features for the derivation is made on the feature diagram itself, and the user can therefore express only one configuration at a time.

For the solution domain models, MODPLFEATUREPLUGIN requires them to be expressed with the FUJABA tool suite. But that is not problematic since the FUJABA tool suite can handle UML models such as UML class diagrams and UML sequence diagrams [FUJ11]. FUJABA UML models are as expressive as others but we cannot import models made via another editor and thus we have to create them in FUJABA even if they already existed but not as a FUJABA model.

MODPLFeaturePlugin is a tool that uses negative variability with direct annotations on the models. It allows the user to remove elements from a complete model, but the user cannot add elements and modify elements. The annotations, which are in FUJABA instances of "UMLTag" can be attached to any element derived from the FUJABA UMLIncrement class. Those elements are every element of the FUJABA model, and for example we can map in a FUJABA UML class diagram elements such as associations, classes, parameters and properties [BuDo09].

The annotations system of this tool allows the user to express mappings between one feature and one element, as well as mappings between several features in an "AND" expression and one element, if the user annotates one element with several annotations. But other expressions such as "OR" expressions cannot be expressed for the mapping.

Adaptability

This tool uses negative variability and thus will have to handle a huge exhaustive UML model if the number of model elements for the product line is huge. Also, using direct annotations brings again more complexity to the same diagram, which then means that models of big product lines will be overloaded and hard to read and use .

If the mapping has to be modified because an element has to be mapped to another feature, it is easy to apply the change. In fact, we only have to change the annotation. Moreover, this tool provides a feature selection which is useful in this case. Also, the tool can detect in a tag a feature that does not belong to the feature model anymore, invalidating the mapping. If the domain solution model has to be changed, the user can apply changes on the model that already has been annotated without having to recreate all the mapping rules.

Usability

This tool provides automatic transformation. The mappings are quite easy to create: it just consists of annotations on model elements that refer to existing features of a feature model. It also provides many functions that improve its usability: for example, when specifying the mapping, the tool provides a way of selecting the features from a list of features that are contained in an existing feature model. It helps avoiding spelling errors and annotating elements with features that don't exist. There is also some visualization techniques that highlights model elements. Additionally, it possesses a model checker that check constraints. This technique is also able to handle propagation rules. Finally, features that are not used in the mapping are noted "unused feature" in the feature model. This tool thus has a good usability, especially on small scale product lines (larger product lines cause scalability problems, which make the approach less usable for that kind of problem).

Maturity

The MODPLFEATUREPLUGIN has been developed recently (since 2009). Such as for AHEAD, the Fujaba tool suite is a mature tool, but the MODPLFEATUREPLUGIN is not. It has only been tested on small scale cases such as in [BuDo092].

4.2.4 CZARNECKI

Expressiveness

Czarnecki's approach has been implemented with the Fmp2rsm eclipse plugin. With this plugin, the feature model has to be described in fmp, and thus we cannot take feature models described with another tool as input. These feature models can express mandatory and optional features, and also AND, XOR and OR decompositions. It is also possible to express feature cardinalities for the OR decomposition. Also, with Fmp2rsm, it is possible to describe all kinds of constraints between features using XPATH expressions. However, they cannot be viewed as a graphical representation. It doesn't provide a way to mark variation points explicitly in the feature model. Finally, it is also possible to express a selection of features in a feature configuration model [CzAn05].

In the approach proposed in [CzAn05], the mapping is said to be able to handle all kinds of model whose metamodel is expressed in Meta Object Facility (MOF). Therefore, it would be possible to use all UML models, which could give a high genericity to the approach. However, with the Fmp2rsm plugin, the solution domain models have to be designed using the Rationale Software Modeler (RSM). This tool can handle all UML models as well, but importing UML models designed with other tools can lead to errors caused by XMI exchange issues.

The template model approach allows the user to express mappings directly on models with a negative variability approach. Like in MODPLFeaturePlugin, the user cannot add elements, but here he is allowed to modify elements by modifying attribute values.

This approach expresses the mapping via presence conditions in order to remove elements and via meta expressions in order to modify elements. But it also allow to express implicit presence conditions in order to express general rules for removing elements depending on their type if they have no presence conditions linked to them.

Also, the presence conditions and meta expressions can be expressed using XPATH, which is a language that allows expressing complex feature expressions. Therefore, as far as the mapping model is concerned, Czarnecki's approach benefits from a good expressiveness (for our personal definition of expressiveness), excepted that it cannot use positive variability.

Adaptability

This approach has the same disadvantages than the MODPLFeaturePlugin approach, since it is also based on direct annotations. Those annotations, in combination with the fact that the base model has to contain every possible element since it is not possible to add new elements while deriving the product model, will overload the model and make it less readable. The coloured visualisation also has scalability issues. When there are a too large number of presence conditions, the colours won't help the visualization because there will be plenty of colours that have only slight differences. According to [Hei09], colours are useful if the numbers of colours used is 12 or less. But in this approach, the scalability issue can be better treated than in MODPLFeaturePlugin thanks to implicit presence conditions, that limits the number of presence conditions to write, and thanks to patches and simplifications that handle automatically the removal of certain elements.

If the mapping should change, this method replaces associated annotations. But this method is again better than MODPLFeaturePlugin because of the implicit presence conditions, patches and simplifications that limit the number of presence conditions, and thus the number of changes to make if the mapping has to be changed. If the domain solution model has to be changed, the user can apply changes on the model that already has been annotated without having to recreate all the mapping rules.

Usability

This approach has been implemented in a tool, Fmp2rsm, in [CzAn05]. This tool provides automatic transformation, but also a visualisation technique that uses colours to highlight the different presence conditions. The mapping is represented as annotations that can be created with XPATH. For the user, it thus requires a bit more effort than the

MODPLFeaturePlugin to learn the syntax of XPATH. Also, this method requires implementing implicit presence conditions, patches and simplifications (which have been made only for the activity diagrams in [CzAn05]. But after having implemented those, using the model templates approach requires less effort than MODPLFeaturePlugin. It thus is a bit more complex to use than MODPLFeaturePlugin for small SPL's, but the patches simplifications and implicit presence conditions make it more suitable for bigger SPL's.

Finally, the tool offers to implement additional processing functions (patches and simplification), which are useful for generating correct and non redundant models. But the implementation of such patches and simplification are not necessary already implemented for every type of model.

Maturity

As for MODPLFeaturePlugin, Fmp2rsm has been tested on small scale cases. The last publication about it last from 2006, and on the official website [fmp11], they state that the plugin, which is a 0.0.5 research prototype version, is no longer developed and could not migrate to RSM 7.0 and 7.5, which can lead to difficulties to make the tool run.

4.2.5 ZIADI

Expressiveness

Ziadi's approach doesn't use a feature model as other approaches do. Instead, it expresses the variability of the product by adding tags on the solution domain models, which is quite similar. Those tags are "optional", "variation" and "variant". So, this method can express variation points explicitly, and can also express OR decomposition of model elements with "variant" tags, as well as optional elements and mandatory elements implicitly (if there is no tag attached to the model element, it is mandatory). Also, the cardinalities of the OR decomposition cannot be expressed, and XOR decompositions cannot be expressed too, whereas we can express AND decomposition by not tagging the elements of a decomposition. Moreover, expressing the variability of the product line on solution domain models rather than with a feature model does not allow the user to express feature relations. For example, if two features, "f1" and "f2" from a product line require each other, it is not possible to specify that every element model mapped to "f1" requires every element model mapped to "f2"(with Ziadi's method, those elements are not explicitly mapped to the features, but they are still mapped in an implicit manner) . But, in this approach, it is still possible to express relations between model elements with OCL constraints [ZJH02] [ZJH031].

Also, the user cannot use a feature configuration model. However, he can express a selection of model elements with the factory models. This lack of feature modelling also has lead the authors of [ZiJe061] to still use a feature model at the beginning of their approach, in order to have a better understanding of the variability in the product line.

In Ziadi's approach, the solution domain model to be used can be of any type. The only constraint is that we must be able to add tags on it. For example, in [ZiJe061] the authors use a UML class diagram and in [ZJH03] a use case diagram. The approach thus has a good genericity.

The approach uses negative variability with direct annotations on models, and produces the derived models based on the annotations and on the derivation algorithm shown in Figure *13*. It is therefore not able to add elements and modify elements.

It doesn't use a feature model, but express the selection of "features" with a factory model (the "decision model") that uses stereotypes to make the link to the annotations of the class diagram. Therefore, in this approach we do not express a mapping from features to model elements but we directly express the selection of elements in a factory. It is then not possible to express feature expressions in derivation rules.

Adaptability

This approach has disadvantages of the negative variability techniques. Also, this technique has to implement a factory for each product variant. The fact that this approach uses factories to allow the user to select what he wants is less scalable than doing this with a feature configuration model, because feature models bring more abstraction to the problem, and for example a feature can be mapped to a large number of model elements, and therefore the selection of features is easier than having to select all the model elements that are optional or variants. It is also less visible, and makes it more difficult to understand the product line than with features.

Also, it creates problems if the mapping has to be changed because the mapping here takes place in every factory. Therefore, a change of the mapping implies changes in a lot of factories. Also, if the user creates a feature model to represent the variability but once has to change it, he will have to apply changes on the model elements, on the models annotations and on the factories as well, and without the help of a tool since the feature model is not linked to the other models in this approach. Finally, a change in the domain solution model will also have repercussions on the factory model. This approach thus has more adaptability problems than the ones that use a feature model.

Usability

This approach is able to automatically derive models based on the model annotations. Otherwise, the usability of this approach is not very good since we have to represent product variants with factories, which is more complex than a variant description model for example. In fact, we have to represent all the model elements where the variation occurs, which are likely to be more than the number of features. Also, using a feature model can help the developers to have a better understanding of the product line than with using the model factories. Thus, the method requires applying refinement on the model in order to specify variation points. We thus have to annotate the model, but also to handle variants ourselves. Finally, compared to other approaches, Ziadi's approach only brings a minimal set of user friendly functionalities.

Maturity

The approach is not mature. It has been tested on a few small scale cases, such as in [ZiJe061] , [ZJH02] and [ZJH031] , and the tool that has been developed is only for research purpose and is not publically available anymore.

4.2.6 FEATUREMAPPER

Expressiveness

FeatureMapper can be used alone, or in combination with Pure::Variants. If used alone, it provides its own feature modelling language, which is shown with Figure 16. This feature mapping language can only organize features in groups, which mean the same as OR decomposition. It thus cannot directly express AND and XOR decompositions, optional and mandatory features, as well as feature cardinalities, feature relations and explicit marking of variation points. It was not able to express a selection of those features until version 0.8.8. It thus has expressiveness problems as far as feature models are concerned.

For the domain solution models, FeatureMapper supports every model as far as it is an ecore-based model. For example, all the UML2 models are usable. It is thus a generic approach, such as mentioned in [Fea11].

As a negative variability tool, FeatureMapper allow the user to remove elements from the solution model domain. With the record function, it can also use positive variability to add elements mapped to a feature. In fact, the recording mode records all changes made on the target model with a selected feature, which can be the removal of elements but also the addition of elements. It uses separate annotation models to define the mapping, and not annotations on the solution domain model itself. The separate annotation model is a set of rules that possess a constraint that indicates the mapped feature(s), and that possess a link to the element models. The fact of having a separate model allows the tool to map elements of several models at a time. With FeatureMapper, we can express AND feature expressions by selecting multiple features from the mapping view. However, other expressions cannot be expressed.

Adaptability

FeatureMapper uses a separate annotation model to express negative variability mappings, and thus has the scalability problem of having to deal with large models in the case of large SPL'S. Also, like Czarnecki's approach, it uses a visualisation technique with colours, which is not very scalable. The fact of having separated the mapping from the solution domain model also brings less complexity to the base model and offers a better scalability than with direct annotations.

If the mapping has to be modified, the required changes are that we have to modify the rules. It can be complex to find an element that we want to change if the number of rules is huge, since they are organised in a tree and there are no find function that we could use to seek these elements or features. Also, modifying a rule created with the record function is not possible, so we have to re-create it from scratch if needed. If the domain solution model or the feature model changes and that, for example, one of their element is removed, FeatureMapper is able to warn the user that a mapping rule is broken because it references a n element that doesn't exist anymore [Hei09].

Usability

The FeatureMapper tool supports automatic transformation. To create mapping, the user as to select a feature in the mapping view and clicking on the related model elements. To create AND feature expressions, the user only has to select multiple features. This is easy for the creation of rules, but less usable when we want to visualize the rules, because we have to reselect the same combination of features to visualize its effects. Also, the derivation with featureMapper can be done from a variant model since version 0.8.8. However, these variant models cannot check all constraints between features. In fact, requires and excludes constraints do not exist with featureMapper. Also, it has functions that improve its usability such as detection of broken mappings and visualisation techniques.

Maturity

FeatureMapper is a tool currently under development (the last version is 0.8.8). It has only been tested on small scale cases, but it is already publically available. Also, some of FeatureMapper documentation is not up-to-date (for example the variants introduced with version 0.8.8 are not mentioned in the documentation because no documentation has been written since then).

4.2.7 FEATUREMAPPER with Pure::Variants

Expressiveness

FeatureMapper can be used alone, or in combination with Pure::Variants. If used with Pure::Variants, it can benefit from Pure::Variants feature modelling language, which is way more expressive and that can also express a selection of features. This feature modelling language contains AND, XOR and OR decompositions, mandatory and optional features, and also a wide range of feature relations (but some of them, such as discourages, are not used by the transformation, but are just there to help the user understand the product line in a better way). Also, it can express variation points explicitly, if the user makes use of the textual description of the features to do so, but there is no graphical representation of that. However, it is not able to express feature cardinalities.

For the domain solution models, FeatureMapper with pure::Variants works similarly to FeatureMapper and also supports every ecore-based model .

FeatureMapper with Pure::variants only offers to remove elements from the model. In fact, it does not import the record function of FeatureMapper. Like FeatureMapper, the rules are specified in a separated annotation model, and can map elements from different target models. Unlike FeatureMapper, it allows the user to create AND and OR Feature expressions, as well as negate a feature in an expression with the NOT reserved word.

Adaptability

FeatureMapper with Pure::Variants uses a separate annotation model and has to deal with large models in the case of large SPL'S, but does not add more information on the base model like direct annotation approaches. It also uses colored visualization techniques, which is not very scalable

If the mapping has to be modified, the required changes are that we have to modify the rules, which can be complex as we have to find the rules in a list without find functions ,

but this list is smaller than the list of features used by FeatureMapper (used alone). Like FeatureMapper, FeatureMapper used with Pure::Variants can warn the user of broken mappings.

Usability

Pure::Variants with FeatureMapper supports automatic transformation, which is made from a variant description model, where features can be selected. Here, creating a mapping is simple and consists of creating rules that are linked to a feature of a feature diagram. For this step the tool provides a way of selecting the feature from a list, which can avoid spelling problems, or making a mapping to a non-existing feature. Then the user can use the graphical interface to click on model elements and add them to the corresponding rule. Also, it has functions that improve its usability such as detection of broken mappings and visualisation techniques.

Maturity

Pure::Variants is an industrial tool and it is used on real life product lines. For the variability mapping problem, it has to be used along with FeatureMapper, which is less mature. and currently under development. The integration of both tools is not documented (there is nothing about it in the Pure::Variants user guide, and there is only one screencast available on Internet).

4.2.8 XSLT with Pure::Variants

Expressiveness

The approach of using XSLT with Pure::Variants requires using a Pure::Variants project that has a feature model as well as a variant description model for each variant. It thus uses the same kind of feature model as the approach of FeatureMapper (if used with Pure::Variants). It can therefore express mandatory and optional features, AND, OR and XOR decompositions, and feature relations, as well as explicit variation points markings with the textual descriptions.

This method maps the features to XML parts. The solution domain model can thus be of any type as long as it is written in XML, such as in the AHEAD approach. So, it benefits from a good genericity.

XSLT is a generic model transformation language. It allows the user to add, remove and modify the elements by allowing making changes in the XML file that models the diagram. Pure::Variants proposes extensions to XSLT [Pur], defined in the namespace “xmlns:pv="http://www.pure-systems.com/purevariants" [Pur09] in order to express directly conditions on the existence of features in a feature selection (the variant description model). Therefore, the user can modify the XML description of a solution domain model, depending on the presence of specific features in a feature configuration model, by for example adding a XSL if clause above the description of the model element. These kinds of clauses also allow the user to express feature expressions.

Adaptability

XSLT used with Pure::Variants can handle positive and negative variability, but a positive variability mapping is more complex to implement. Thus, if the user doesn't want to handle the complexity of the positive variability mapping, he has to handle the problem of having a big base model if the SPL is large. If it uses positive variability, then it offers a good scalability since he can have a smaller solution domain model and also parse feature models and take actions based on the type and attributes of feature model elements.

If the mapping needs to be changed, there would be no impact on the script if it is generic enough. But that is unlikely and, if it is not, the user would have to find the features where the mapping has to be changed and implement the changes. If the feature model or the solution domain model evolves and that a one of their element changes, there is no checker tool to warn the user that a mapping is broken, and this can lead to transformation errors. Also, because we treat the solution domain as a XML file, an evolution of this model (for example in a graphical editor) implies that the user has to apply changes manually on the XML description of the model in the XSLT file, or recreate a whole new mapping.

Usability

As discussed above, using XSLT for expressing a positive variability mapping is very complex because we have to deal with low-level XML details while composing the XML document model. But the negative approach can make use of an already complete domain model (and thus we don't have to write XML parts, but just remove some of them). It also makes use of generic actions (actions based on types or attributes of features), which are easy to implement and can avoid us to create mapping rules for every feature, which improves the usability of the tool when facing large mappings. Also, it provides an automatic transformation. However, there is little tool support for editing the XSLT script. In fact, the editor doesn't provide spelling check or existence check at the time of the editing. The only support function is the fact that the transformation warns us when the script is not syntactically correct, or when features specified have not been found, or when the script contains never ending loops.

Maturity

XSLT is a mature low-level XML transformation technique developed by the World Wide Web Consortium (W3C) and the most current version is XSLT 2.0, which was released in 2007. As discussed above, we use an extension of this language developed by Pure Systems, and available with Pure::Variants, which is also a mature industrial tool. However, Pure::Variants XSLT was not made for the purpose of editing solution domain models, and there is no documentation or case studies on that problem.

4.2.9 CVL

Expressiveness

CVL does not use a feature model to express the variability of the product line. Instead, it expresses it in the CVL model with the "CVL elements" that are shown in Table 2 [**Fle09**].

Semantics	Symbol	CVL Element	Comment
Mandatory		:CompVar	*CompositeVariability* makes the resolution of a node mandatory for any instantiation
Optional		:Iterator Lower =0 Upper = 1	The multiplicity [0..1] reflect the fact that the sub-node can be chosen or not.
AND		:CompVar	CompositeVariability makes the resolution of all sub-nodes mandatory for any instantiation
OR		:Iterator **Lower =1** **Upper = -1** **IsUnique = true**	The multiplicity [0..*] means that any number of sub-nodes can be chosen. *IsUnique* specifies that each sub-node can only be chosen once (which corresponds to the usual OR semantics)
XOR		:Iterator Lower =1 Upper = 1	The multiplicity [1..1] means that one and only one of the sub-nodes can be chosen (which corresponds to a classic XOR semantics).
Multiplicity		:Iterator **Lower = x** **Upper = y** **IsUnique = true**	The multiplicities associated to a choice are mapped to an Iterator containing the same multiplicities

Table 2: CVL model elements [Fle09**]**

With the elements "CompVar" and "Iterator", we can express in the CVL model mandatory and optional features, AND, OR and XOR decomposition and feature cardinalities (here called "multiplicity"). It is however not possible to express relations between features and to mark variation points explicitly. CVL also uses its own way of describing variants, with the resolution model. The choice of defining a few general concepts such as the Iterator, rather than reusing an existing feature diagram metamodel with a metaclass for every symbol, is "to remain general and concise and be able to support various feature diagram notations" [CVL]. But, in [Fle09], they still use a standard feature model (expressed in FODA) to express the variability of the product line, showing that expressing the variability in the CVL model does not give an overview of the SPL as good as expressing it with a feature model that would be able to express directly every concept that we listed for our expressiveness criterion.

The solution domain model, called base model in CVL, can be of any type as CVL is a "generic approach in the sense that the supported transformations work on any model in any language" [Fle09].In fact, the goal of CVL is to offer a common variability language that can map features to elements of any "domain specific language", or DSL. It is thus a highly generic approach.

In CVL, the transformation is based on aspect oriented modelling techniques such as substitutions. It allows the user to remove elements from a model, to add elements and to modify elements thanks to the substitutions. Those substitutions are used in the "product realisation layer", to define the mapping between the feature specification and model elements via transformation operations, based on the three operators (value substitution, reference substitution and fragment substitution). Those operations can concern every kind of elements.

Adaptability

CVL allows the user to use the three variability mechanisms. Therefore, for large SPL's the user can choose a positive variability approach (such as with VML*) to avoid scalability problems linked to the negative variability. In fact, [Fle09] points out that "CVL does not require the union of all products to be available as model elements in one model, as e.g. FeatureMapper or PureVariants' Enterprise Architect integration". Also, the approach of CVL is based on aspect oriented modelling, thus when it uses positive variability it encapsulates elements into one aspect separately from the other aspects. This kind of decomposition improves modularization, which "eases maintenance and evolution" [CVL]. But, this approach uses a graphical editor to present the CVL model, which contains the variability of the product line but also references the substitutions, such as shown in Figure 17. When facing large SPL's, this model can thus become overloaded and difficult to read.

If the mapping has evolves, the user can benefit from the decomposition of the elements into aspects, which eases this evolution. But, if the feature model evolves because of a change in the product line, the user also has to apply those changes in the CVL model. Finally, if the syntax of the solution domain model changes, it can still be used with CVL, but the user has to re-implement the CVL API that allows to view the model, and the previously specified mapping is lost.

Usability

CVL provides an automatic way of transforming product models with its model transformation abilities. In CVL, using the simple constructs of CVL (the simple substitutions) efficiently is straightforward. Using the more advanced concepts requires more detailed knowledge of CVL [CVL]. However, CVL requires expressing the connections to the target model language and how to apply transformations on it. It thus implies an additional cost in time and effort when we want to use it on a new language. For example, in [Fle09], they redefined the concrete syntax of the CVL model as a combination of CVL and TCL. CVL does bring a default concrete syntax for this model, but it can be changed to "better fit the target DSL" [Fle09]. Also, CVL provides tool support such as checking the validity of the model regarding to its meta-model, a graphical editor that allows to modify the CVL model but also select and highlight elements and add them into aspects.

Maturity

CVL is a tool (an Eclipse plugin) under development. It has been tested on small scale cases, for example for the DSL "TCL" (train control language) in [Fle09] . The last version is version 1.2 and has been released in 2009.

4.2.10 VML*

Expressiveness

With VML*, the feature model can be of any type, because the user has to enter information about the feature modelling language metamodel in the VML language description file [Zsc091]. For example, VML4RE uses feature models expressed in FMP [Amp] (which is also used by fmp2rsm, the plugin from czarnecki's approach). The expressiveness thus depends on the choice of the user when he creates the description file, or

when he decides to re-use an already defined description file. As discussed above, the feature models from FMP can contain mandatory and optional features, and also AND, XOR and OR decompositions, feature cardinalities, and constraints between features. Also, VML* languages implement a "configuration import" [Zsc09]that is used as an adapter for loading configurations stored in models. Thanks to this configuration it can automatically generate product models depending on the selection of features and the actions described in the rules.

The solution domain models used can be of different types. The target model just has to be an EMF based model. As for the feature model, the user has to enter information about the solution domain model meta model in the language description file. Such as written in [Hei09], the approach is generically applicable for any EMF model, but the drawback is that there are additional costs at the beginning of a project, if the language description file has to be written.

VML* is a generic approach to the variability mapping problem that allows to implement customised languages that will do the mapping for a special type of target model. Those languages have the possibility of being able to add, remove and modify elements from a model. Also, within the VML language description, it is possible to define actions that map features to any type of model element of the solution domain model. Also, according to [Hei09], the actions that can be implemented can be much more powerful than generic direct mappings between features and model elements. Finally, the mappings are defined as rules that are named "variants". They allow expressing feature expressions.

Adaptability

VML* allows the user to use the three variability mechanisms, and thus he can choose the mechanisms that are the most appropriate for every situation, depending on the fact that he don't want to deal with large model, or the fact that he wants to have a better overview of the SPL. The scalability of the approach also depends on the created mapping language, which will define the different possible actions. But it is still possible to refine a mapping language after having used it. Also, because the user can define its own actions, these actions can be more adapted to large scale product lines than generic direct mappings such as in other approaches, where the user has to define the mapping for each feature and/ or feature expression. With VML*, the user can also re-use actions that he has defined in other actions.

If the mapping has to be modified, then the user has to find the corresponding feature or model element in the text file. Also, if the feature model or the solution domain model evolves, the tool can detect the changes with its broken mapping detector. Therefore, the evolution of a model is does not cause problems to the VML* user.

Usability

In VML*, the VML language description contains all the required for automatically deriving the architectural model of a specific product from the reference architecture [FNS09]. But this language have to be written by the user (or he can also re-use an existing one) and therefore, there can be additional initial costs. This is tool does not bring support such as graphical editors and (direct) visualisation techniques. Compared to approaches that use graphical representations such as FeatureMapper, VML* is less good at helping the user to have a good overview of the SPL [Hei09]. But it is able to generate trace links [Zsc091] that

relate feature to added, modified and removed elements and that can be visualised using the AMPLE traceability framework. Those traceability links are helpful for "discovering candidates of bad features interactions, visualizing variations in different requirements models and are also valuable for to analyze the design change impact when evolving SPL [Alf09]. VML* also provides a model checker ability that checks if a feature referenced in an expression actually exist in the feature model, and a broken mapping detector. Also, if the user makes use of positive and negative variability at the same time, this could lead to problems, for example if he would ask to remove a model element that should have been added with another action, but which would not have been executed already. The order of the actions can be defined in VML* but it is still a source of possible errors. The mapping is represented in VML* as code lines. In [Hei09], they found that the average amount of code lines for an action implementation of the VML4ARCH language is about 15 lines. After being implemented, it takes one code line to use the implemented action. It is then quite usable, but not as much as a declarative annotation.

Maturity

VML* has been tested on small scale product line such as in [Alf09], where they use the VML4RE language specification. The approach is implemented in a tool that is publically available, and the VML4RE and VML4ARCH languages are also available.

4.3 Conclusion of the literature-based comparison

In the last section we compared the different tools by applying the criteria that we had defined before onto them. Now, we will summarize the main facts that we discovered during this comparison, for the expressiveness, adaptability, usability and maturity of those tools. The table below shows our evaluation of the different tools for each criterion.

	FLOCOS GPL	AHEAD	MODPLFEATUREPLUGIN (Fujaba)	Fmp2rsm (Czarnecki)	ZIADI	FEATURE MAPPER	PURE :: VARIANTS WITH FEATURE MAPPER	XSLT	CVL	VML*
EXPRESSIVENESS	-	+-	+-	+	+-	+-	+	+	++	+++
ADAPTABILITY	---	+	-	+-	--	+	+	++	+++	+++
USABILITY	--	-	+	++	-	+	++	-	+	+-
MATURITY	--	-	-	-	-	+-	+	+-	+	+

Table 3 : comparative evaluation of the different tools

The different tools and approaches that we have analysed are difficult to rank depending on their expressiveness, because some can express things that other cannot, and vice-versa. For the expressiveness, we think that the most important criterion is the expressiveness of the mapping, while the ability of handling good feature models and a wide range of target models is a bit less important. Also, the explicit marking of variation points is a less important criterion than the others. So, in general, we can say that XSLT, CVL and VML* have a better expressiveness than the other approaches. In fact, they can handle positive and negative variability as well as element modification mechanisms, whereas the others approaches cannot handle positive and negative techniques at the same time (excepted for FeatureMapper, which can use positive variability with the record function). Moreover, only one declarative approach that uses negative declarativity, the approach of Czarnecki, was able to modify elements. But, in the other hand, mixing the fact of adding and removing elements may cause problems in product derivations [Hei09]: "when negative variability is used to remove a model element from a larger model fragment that is added to the core model using positive variability, it is important that the model fragment is added before the negative-variability remove actions are executed". Also, we saw that VML* and CVL could handle positive variability in a better way than XSLT. Also, they are customizable for a lot of modelling languages. But VML* is compatible to every feature modelling language whereas CVL uses its own variability modelling language. Thus we could say that VML*, and then CVL, are the most expressive techniques. On the contrary, the FLOCOSGPL approach, even if it can handle all types of feature and target models, can only express the mappings with a simple table. We thus evaluated it as the less expressive one. The expressivity of AHEAD was not very good too, because of its lack of feature representation. Declarative techniques are thus in general less expressive than operative ones. For those techniques, the approach of Czarnecki seemed to be the most expressive, as it is able to express the modification of elements and feature expressions, as well as patches and simplifications. The combination of Pure::Variants and FeatureMapper provided a good expressiveness for a declarative technique as well.

Concerning the adaptability criterion, we saw that positive variability brings less scalability problems than negative variability. But because they both raise different problems, the best solution is to be able to use both techniques and to choose the most appropriate one, or both, in each case. Thus, VML* and CVL seem to bring more scalability than the other techniques. Also, we saw that using colours for visualisations was not very scalable. For the negative declarative techniques, we saw that FeatureMapper was more scalable because having the mapping in a separate model can avoid bringing more complexity to the models. A separate model is also more efficient when one of the other models has to evolve. For the direct annotation techniques, the approach given by Czarnecki was the best one because it can handle several mappings implicitly. Also, the method of FLOCOSGPL was the worst as far as adaptability is concerned, and the method of Ziadi also had adaptability problems because it uses tags on target models and factories to express the mappings.

As we analysed the usability of the different techniques, we found that declarative variability mapping techniques that use negative variability were more usable because annotations on the model or on a separate model are really easy to create. VML and CVL require some additional effort and time at the beginning in order to specify how the

transformations are going to be made. In the declarative model transformation approaches, this is already implemented and thus we can use the transformation tool in less time. Also, we saw that XSLT and VML are based on text editors, and thus the way we use them is not as easy as other approaches that offers graphical editors. However, these graphical approaches tend to be less usable as the size of the product line increase, but it is also true for text editors, to a lesser extent. XSLT and AHEAD also had the same problem of having no representation of the target model. These methods base themselves on XML document transformations, but this brings less visibility to the user, and he can also create transformations that will lead the generated model to be invalid. In fact, in [TBD07], they argue that " the mapping is less easy to create (in AHEAD) than in other approaches like FeatureMapper, because with AHEAD we have to handle the composition of the parts as well as the XML based refinements, whereas with tools like FeatureMapper the mapping is created using the graphical user interface by clicking on model elements». But, for large SPL's, or when we intend to reuse the transformation for many SPL's, the operational techniques (CVL, VML* and XSLT) become more usable as declarative methods tend to have scalability problems and thus become less usable. Finally, some of the methods that we analysed were only a definition of an approach to the problem, and were thus quite unusable, such as FLOCOSGPL.

None of the different studied approaches were perfectly mature. If Pure::Variants is a mature tool, the combination with FeatureMapper is not mature yet because FeatureMapper is still in development. And for XSLT, this technique is not focused on the variability problem, and therefore there is no documentation neither case studies that were made for creating variability mappings with XSLT. For AHEAD, we had the same problem of combining a not mature plugin to a more mature environment, but also that approach already is a bit outdated and doesn't seem to be developed anymore. The approach of Czarnecki has also been discontinued, due to compatibility problems. Compared to this, the MODPLFeaturePlugin is a recent project which is still in development. The same goes for CVL and VML*, but for them we have found more documentation and case studys. Finally, the approach of FLOCOSGPL has never been implemented because it is more a method than a tool.

5 Empirical comparison of the tools

In chapter four we presented a literature-based comparison of the different approaches of the variability mapping problem that we introduced in chapter three. In order to complete this comparison and benefit from another point of view, we will in this chapter make an empirical comparison of the tools. We will first introduce the comparison criteria that will be applied on the tools. Then, we will present the case study that will have to be implemented by the tools, and after that we will compare how the different approaches managed to implement the case study, based on our criteria. Because not all of the tools were available, we couldn't include all of them in the empirical comparison. Finally, we will end this chapter with a conclusion of the empirical comparison.

5.1 Comparison criteria

In order to make a comparison of the different studied tools, we will first define the different criteria that we will take into account when evaluating those tools based on our application of the case study (presented in the next section) to them.

5.1.1 Usability

Such as for the literature-based analysis, we will analyse the usability of the different tools for the empirical comparison. Here, we will evaluate the estimated effort for creating a mapping. To do so, we will estimate the average number of actions needed to create a certain element of the mapping, which will be an average of the number of lines of code, or an average of the number of clicks, such as made in [Hei09]. For every tool, the element studied will be the creation of the same mapping rule. We will also analyse the effort needed for tool-specific actions (if any exist), and the total amount of code lines or mapping model elements needed, in order to have an overview of the total estimated effort required with each tool.

In the usability criterion, we will also debate about the usefulness of the support functionalities that the different tools bring, such as code completion, feature highlighting,…, which can make a tool more easy to use.

5.1.2 Expressiveness

With this criterion, we will evaluate if each tool was able to express every concept that the case study (introduced in the next section) contains. It will thus complete the literature-based analysis of the expressiveness, with in this case a focus on the concepts that are likely to be used in a "real life SPL".

5.1.3 Performance

This criterion evaluates the performance of the tool, such as proposed in [MKR06]. We will thus evaluate the time needed to perform the model transformation, if that feature exists. That evaluation will be performed on the same environment, in the same conditions for each tool, in order to ensure the correctness of the relation between the results.

5.2 Case study presentation

In order to compare the different approaches with equity, we need one standard case study that will be used for all of them. So, this section introduces the case study that we will use in the following of this chapter.

The case study that we will use is the case of the software product line named REACT. It is a software product line that has been engineered at University of Luxembourg, and that describe different products that belong in the scope of crisis management solutions. We also used this case in [Lec11]. The case is composed of a feature model describing the variability of the REACT product line, one UML class diagram named "businessObjects" that represents architectural elements of the product line and the mapping from the features to the model elements.

The feature model of REACT is shown in annex A. Its features are decomposed into five groups: Resources, Goals, Processes, Dependability and Documentation. The resources group contains features that define human resources and material resources needed to handle a specific type of crisis, such as "ArmySpecialUnit", or "FirstAidMaterial". The Goals group contains features that define the main goal of the product, divided into three subgroups (area size, missions to handle and crisis type), and therefore contains features such as "NurseTheWounded", "Small", or "Earthquake". The Processes group contains features that define how the product will handle the crisis, decomposed into subgroups for each subprocess (SignalDetectionProcess, PreparationProcess,controlProcess,…). It thus contains features such as SensorDetection or VideoAnalysis. The dependability group contains features that define the level of security, availability, reliability and confidentiality of the system. Finally, the documentation group contains features that define which type of documentation will be attached to the product (user guide, development documents,…)

The different constraints on the features of the product line are constraints such as:

- Worker requires communicationDevice
- GSM requires GSMProtocols
- PDA requires VPN OR Internet OR PrivateWireless
- WarningServiceSubscriber requires PublishInfoForSubscribers, Recommends TrackingTechnology AND Track
- Small(area) recommends VideoSurveillanceSystem, Discourages GPS AND PrivateWireless
- Large(area) recommends Mulitcoordinators AND PublicHospital AND FlyingItem AND Transport, discourages Walkietalkie AND VideoSurveillanceSystem
- RemoveObstacle recommends GarageTowTruck
- NurseTheWounded requires MedicalServices OR FirstAidWorker
- NaturalDisasters requires FireDepartment AND ExternalCompany
- PlantExplosion Requires FireDepartment AND PublicHospital AND Police
- NuclearPlantExplosion requires ArmySpecialUnit
- MajorAccident requires FireDepartment AND PublicHospital AND PrivateAmbulanceCompany AND PrivateHospital AND Police AND IndependantFirstAidDoctor

- Logging requires AuthenticationSystem
- VideoAnalysis requires VideoSurveillanceSystem
- SensorDetection requires Sensors
- AuthorizationSystem requires AuthenticationSystem AND (VPN or Internet OR privateWireless)
- AutorizationSystem discourages TalkieWalkieProtocols AND GSMProtocols
- AuthenticationSystem discourages Walkitalkie
- Send conlicts (=excludes) Publish
- HighSecurity requires DataEncrytion

For space concern, we did not list all the constraints. Since the case study was first designed using Pure::Variants in [Lec11], it contains special constraints such as "recommends" and "discourages" Also, the constraints "discourages" and "recommends" does not affect the transformation process of the product. A discouraged feature can in fact be selected for a product.

The domain solution model called BusinessObjects is shown in annex B[5]. It contains the different business objects of the architecture of the product line, which have to be selected for some products derived from the SPL. It contains classes such as the different missions that can be assigned to a worker (it contains the superclass "Mission", as well as subclasses "MissionPump", "MissiontravelTo", MissionRescue",…), the different human resources needed, decomposed into internal and external human resources (for example, a possible internal human resource is "Coordinator", and a possible external human resource is "MedicalHumanresoucre"), and other resources such as "FirstAidMaterial" or "CommunicationDevice".

The mapping that has to be implemented by the tools is mainly composed of 1 to 1 mappings such as:

- Feature Pumping -> class "MissionPump"
- Feature FirstAidWorker -> Class " FirstAidWorker"

But it also contains more sophisticated mappings. For example, one mapping makes the link between the feature expression " InternalHumanResource OR ExternalResources" and the class "HumanResource" and also to "User" (and also the associations between "User", "MobileDevice" and "user"). The mapping is quite simple because the relations between the elements are implemented in the feature level. In fact, the mission observe requires an observer, but since the feature Observe requires the feature Observer (in the REACT case study implemented with Pure::variants and featureMapper), there is no need to implement that in the mapping level.

There is no special attribute to map in this case study. As far as the association links are concerned, they can be mapped by following rules: if it is an implementation link from a

[5] This model shows the architectural elements of the REACT spl as it was on January 2011, and contains a few differences regarding what is expressed in the feature model of the REACT SPL (for example, some missions defined in the feature model were not already implemented in the product line and thus are not represented in the class diagram).

subclass to a superclass, the link has to be in the same mapping rule as the subclass. If it is an association link between an optional and a mandatory class (i.e. a class that belongs to the core elements of the model), the link has to be in the same mapping rule as the optional class. If it is a link between two optional classes but with one class that depends on the other (for example because of a “requires” constraint), the link has to be in the same mapping rule as the depending class. Finally, if there is no such dependency, we have to create a AND feature expression of those classes and map the link to this expression.

The mapping rule that will be used for the usability criterion is the one that maps the feature Observer to the class Observer, its association link “CreatedBy” to the class Mission and its generalization link to the class AidWorker.

5.3 Comparison of the tools

Now that we have defined the comparison criteria and the case study, we will apply them on the tools that were available.

5.3.1 FlocosGPL

The approach of FlocosGPL is not supported by a tool. There are no constraints on the feature model and the target model. So, when applying the case study to this approach, all we had to do is to create the table below that defines the mapping. Each of its line contains one feature that is mapped to one or more model element. The feature model that we used was the one from the case study (defined with Pure::Variants). The target model used was the one from the case study as well. Then, because the approach does not bring automatic transformation, we have to remove manually from the base model all the elements that are listed in the table and linked to a non-selected feature.

Feature	**Model elements**
Observe	MissionObserve
Observer	Observer
FirstAidMaterial	FirstAidMaterial
MedicalResource	MedicalHumanResource
GovernmentalServices	GovernmentalHumanResource
Worker	AidWorker
ExternalResources	ExternalHumanResource, HumanResource
InternalHumanResource	InternalHumanResource, HumanResource
SystemAdmin	Admin
Manager	Manager
InternalHumanResource	HumanResource
RemoveObstacle	MissionRemoveObstacle
Pumping	MissionPump
Witness	Witness
Missions	Mission
FirstAidWorker	FirstAidWorker

Coordinator	Coordinator
Logistician	Logistician
Rescue	MissionRescue
CommunicationDevices	MobileDevice
CallCenterEmployee	CallCenterEmployee
Transport	MissionTransport
MedicalServices	MedicalHumanResource
MaterialResource	Materialresource
TravelTo	MissionTravelTo

Table 4 : Mapping of the REACT SPL with the FLOCOSGPL approach

Expressiveness

As we could re-use the feature diagram and the target model from the case study, there were no expressiveness problems for that point. But in general, the expressiveness of this approach depends on the models used.

For the mapping, we were not able to express every concepts of it with the mapping table. In fact, if the 1-1 mappings didn't cause problems, the approach cannot express feature expressions. So, we had to decompose the feature expression "internalHumanResource OR ExternalResources" into two different mapping rules that map to the same element (Humanresource). Also, for the model elements, we specified classes only, as the approach isn't supposed to handle other elements.

Usability

In order to create the Observer rule, we followed these steps:

- Create the line with the feature "Observer"
- Add the "Observer class name in the model element column.

The mapping rule is thus easily created, but we couldn't add the association links and generalization links to the mapping, as the approach is not supposed to support that.

Otherwise, the usability of FLOCOSGPL approach is the worst of all the tested approaches because it is the only one that does not bring automatic transformation, and so we had to do it ourselves while applying the case study. Moreover, there is no tool implemented for the approach, which means no visualization technique available, and we cannot use more than one target model at a time.

Performance

This criterion is obviously not relevant for this approach.

5.3.2 AHEAD

The AHEAD tool suite and its plugin XAK provide a mean of composing XML documents. It is meant to be used with command line instructions, and there is no graphical user interface for it. So, the first step to do while applying this approach onto the case study is to get the XML source code of the whole target model. Since AHEAD does not use feature models and mapping models, we also need to acquire a feature model and the mapping rules from somewhere else in order to have an overview of the variability of the REACT product line and of the mapping to realize. So, we took the feature model and the mapping model directly from the case study. Then, we had to define the core model, and so we created a new XML document copied from the original XML source code of the target model, from which we removed every "packaged element" that was referencing a model element mapped to an optional feature. Then, for every mapping rule, we had to create a new XML file that contains the code of the optional model elements mapped to the rule, and that defines the XAK code of the refinement (see below for the mapping rule Observer example). When all these documents are created, we are able to process a model transformation. The approach of AHEAD doesn't use a variant model to do so. In fact, a transformation is specified by writing the following command lines in a command prompt, assuming that the core model XML file is named core.xml, and output.uml is the output file:

Xak –c core.xml feature1.xml feature2.xml –o output.uml

This command corresponds to the application of the refinement feature1 and then of the refinement feature2 on the core model.

Expressiveness

The approach of AHEAD does not use any kind of feature modeling language, but relies on the application of refinement files on base model files. So, it is impossible to express the concepts of a feature model and of a variant model with AHEAD. For the target model, we were able to use the one from the case study, since it is written in XML.

For the mapping, AHEAD doesn't rely on a model but the different mapping rules are represented by the different refinement files. These refinements can handle positive and negative variability with the xak actions "append" "prepend" and "override". Also, there is no constraint on the contents of the file, so it can be a refinement for a 1-1 mapping rule, but also for a mapping rule that maps a feature expression. The refinements can aim every type of model element, because they handle low-level XML code.

Usability

In order to create the mapping rule Observer with AHEAD, we have to do the following steps:

- Create a new file observer.xml
- Write the following in the file:

<xr:refine>

```
 <xr:at select="//uml:Package[@name='lu.uni.lassy.react.server.bo']">
  <xr:prepend>
//XML of the observer packaged element
<packagedElement xmi:type="uml:Class" xmi:id="_h3X7wDaJEd-T3YEG-mJywA"
name="Observer">
<generalization xmi:id="_upimcDwKEd-VwaPrEsLlQw" general="_TThzgDWnEd-0G4fN-jL5IQ"/>
<ownedAttribute xmi:id="_j4xLwDaJEd-T3YEG-mJywA" name="ROLE" isStatic="true"
isReadOnly="true" aggregation="composite">
<type xmi:type="uml:PrimitiveType"
href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
<defaultValue xmi:type="uml:LiteralString" xmi:id="_nteMcDaJEd-T3YEG-mJywA"
value="Observer"/>
  </ownedAttribute>
(….) // rest of the XML of the observer packaged element
</packagedElement>
</xr:prepend>
 </xr:at>
 <xr:at select="//uml:Package[@name='lu.uni.lassy.react.server.bo']">
  <xr:append>
   //XML of the OBSERVER association
<packagedElement xmi:type="uml:Association" xmi:id="_iOndQE7WEd-_coWA0tzTLg"
name="createdby" visibility="private" memberEnd="_iOndQU7WEd-_coWA0tzTLg
_iOoEUk7WEd-_coWA0tzTLg">
<ownedEnd xmi:id="_iOndQU7WEd-_coWA0tzTLg" name="src" type="_1yqZwDZuEd-
gUK_7ELsHLg" association="_iOndQE7WEd-_coWA0tzTLg">
<upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_iOoEUU7WEd-_coWA0tzTLg"
value="1"/>
<lowerValue xmi:type="uml:LiteralInteger" xmi:id="_iOoEUE7WEd-_coWA0tzTLg" value="1"/>
  </ownedEnd>
  <ownedEnd xmi:id="_iOoEUk7WEd-_coWA0tzTLg" name="dst" type="_h3X7wDaJEd-T3YEG-
mJywA" association="_iOndQE7WEd-_coWA0tzTLg">
<upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_iOorYU7WEd-_coWA0tzTLg"
value="1"/>
<lowerValue xmi:type="uml:LiteralInteger" xmi:id="_iOorYE7WEd-_coWA0tzTLg" value="1"/>
  </ownedEnd>
 </packagedElement>
  </xr:append>
 </xr:at>
</xr:refine>
```

So, the mapping rule created with AHEAD takes only two steps, but the second step is very complex because the user has to write the XML code of the refinement elements within a refinement function, and also specify where it has to be applied in the base model file. Also, the user has to include the associations,… himself in the mapping, which can be not easy to find in a large xml document. Moreover, the user has to pay attention that he generates a valid model. In fact, with other approaches, there are some constraints to respect, for example on the selection of the features for a variant. Here, the user can do everything he wants for the mapping since there are no feature model, and also because he handles low-level XML. For

example, it is possible to add the class admin to the refined model, and its generalization link to aidWorker, but not its superclass AidWorker, which will generate an invalid model. A user is also likely to forget a link, and will not be warned for this.

Also, the mapping with AHEAD requires to get a full model of the product line, and then remove the optional elements in order to create a core model. In fact, it would be very hard to start directly with a core model and write the XML parts for the refinements ourselves. It is thus more costly in terms of actions at the beginning of the process than with some other approaches. Having to handle XML code also means that the user has to handle a quite large file (more than 1800 lines of code for our case study) which is clearly less usable than handling models in a graphical user interface.

As we discussed above, it is impossible to create a variant with AHEAD. In fact, we want to process a transformation, we have to specify all the refinements that we want to apply. So, for example, in our case study, generating the whole model with every possible element would require to write a command that specifies the 26 different refinement files. So, it is a big lack of usability because we cannot ensure that variants are valid, because we cannot save them, and because specifying them in the command line is not very user-friendly. It also requires handling a large number of different files. Moreover, the order in which they are specified determines the order in which they are applied. This could be dangerous for example if a refinement is supposed to remove one element that would have been added before, or link an element to one that would have been added. In fact, if the refinement that would have added that element was not applied before, the transformation will be invalid.

Finally, another big lack of usability of AHEAD comes from the fact that it does not bring a graphical user interface. So, unlike other approaches, here the user cannot have a visualization of the effects of a mapping on the target model, or see which elements are linked to which rule (here, it would be a file).

Performance

Performing the transformation with AHEAD took less than ten seconds, which is still less than the time needed to write the command line.

5.3.3 MODPLFEATUREPLUGIN

This tool is no longer under development and not available anymore. Consequently we are not able to include it in the comparison.

5.3.4 FMP2RSM (Czarnecki's approach)

This tool is also no longer under development. It requires the IBM Rational Software Modeler version 6.0.1.1 which is no longer available, and so we cannot include this approach in the comparison.

5.3.5 Ziadi

The tool developed for this approach is, like the two previous ones, unavailable, and so we cannot include it in the empirical comparison.

5.3.6 FeatureMapper

The first step to do with FeatureMapper (without Pure::Variants), while applying this case study, is to re-implement the feature model as a FeatureMapper feature model (found in "EMF model creation Wizard" in Eclipse), because it is the only feature model type supported. As we will explain in the expressiveness subsection, it is not possible to express some concepts of the case study such as constraints, but it is still enough to allow a valid transformation. After that, we have to import the business objects class diagram. Here, the class diagram from REACT is directly importable, since FeatureMapper is able to handle all ecore based models. Then, we create the mapping file, which is a specific .featuremapping file defined by FeatureMapper. We have to specify the target model (multiple target models are possible) and the feature model that will be used by the mapping file. Then, in the FeatureMapper mapping view, we add, for each feature or feature expression created, the related model elements by clicking on them in the graphical editor after having selected a feature or feature expression in the mapping view, and then clicking on the apply term button. Doing this step for every optional model element makes the mapping complete.

After that, with FeatureMapper 0.8.7, in order to derive a class diagram for a specific product, we have to copy the feature model and rename it, and then deleting from this copy all unwanted features. Then, we click on the transform button, select the copied feature model, the output folder and FeatureMapper automatically derives the class diagram. Since FeatureMapper 0.8.8, defining variant models is supported. We can then make a selection of feature via these models, and then use them as input for the derivation.

Expressiveness

As we implemented the REACT case study, we were not able to express some of its concepts. Firstly, the feature model used by FeatureMapper did not allow us to define the constraints between the features. Secondly, this model uses cardinalities to express concepts such as mandatory features. So, for a mandatory feature, we defined its cardinalities as 1-1, such as for the feature "Goals" from Figure 20. For optional features, we defined their cardinalities as 0-1. For a group of alternative features, we used 0-1 cardinalities in a feature group with 1-1 cardinality. AND groups are defined as a group of mandatory features with the correct cardinalities (for example, the group composed of the features "Area", "Missions" and "Crisistype" has 3-3 cardinalities) and OR groups are defined with optional features with a minimum cardinality equals to 1 for the group.

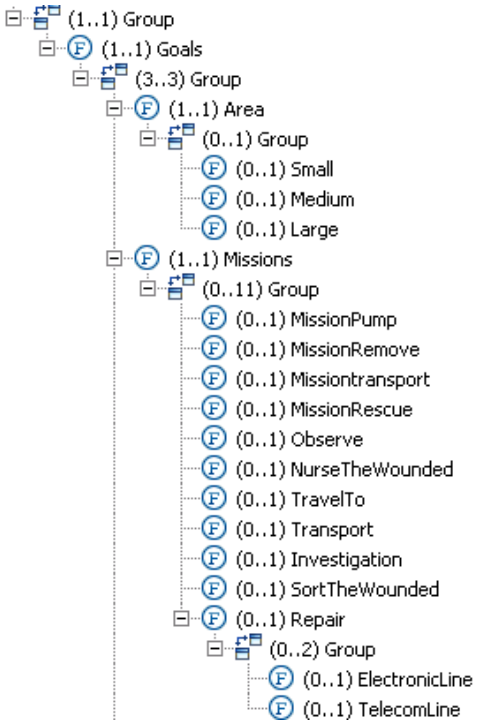


Figure 20: Part of the REACT feature model expressed via FeatureMapper

Also, we can see on Figure 20 that groups cannot be named, and that features cannot have children features directly but only groups of features, because it is required in order to specify the cardinalities, and therefore to express AND and OR decompositions. Finally, the version 0.8.8 of FeatureMapper integrates a mean of expressing variant models.

If the feature model has a few lack of expressiveness, the mapping model of Featuremapper allowed us to express all the mapping rules of the case study without problem. In fact, we were able to map classes but also association and generalization links, as FeatureMapper is able to map every model element. Also, we were able to express AND feature expressions. We could not express OR feature expressions directly, but we handled this problem by creating to different rules with different related features but with the same model elements. Finally, we were able to import the original solution model domain.

Usability

With FeatureMapper, the mappings are created quite easily. The steps needed to create the rule « Observer » are :

- Select “Observer” in the feature list in the mapping view
- Select the Observer class
- Add the association link “createdby” to the selection of model elements
- Add the generalization link to AidWorker the selection of model elements
- Click on “Apply term”

This mapping rule thus requires 5 simple actions, made in 5 clicks. It doesn’t take long to define the mapping of the REACT SPL composed of 26 mapping rules. It is also helpful to benefit from the different visualization techniques to see what elements are already mapped,

and what would a model derived from a variant look like with the variant filter. Also, the mapping allows to use several target models, which avoid us to redefine the mapping rules for each target model (if we use more than one). In order to create an AND Feature expression, we only have to select multiple features, and the expression is created automatically.

But we also found that the feature model and its derived variant model cause a few usability problems. Firstly, using cardinalities for making the difference between the type of feature and decompositions is less usable than doing it with icons, like Pure::variants does. In fact, it is less easy to spot the type of one feature directly, because we have to check the cardinalities of the feature and of the group it belongs to instead of seeing it directly with an icon, and also, it is less easy to create (for example, for each group, we have to compute its maximum cardinality, and for every child feature of this group, wet have to set the cardinalities manually). We found that the feature modeling of FeatureMapper was more like an input for the transformation than a real mean of expressing the variability.

The variant model brings some checking ability, but not for every kind of feature. We found that it was able to detect unselected mandatory features and the improper selection of two or more alternative features when they belong to a group with 1-1 cardinalities, but it was not able to detect the violation of every other cardinalities (for example, a group with 2-4 cardinalities) and also the non selection of a mandatory feature (with 1-1 cardinalities) in a group a features with cardinalities different than 1-1. Also, another usability drawback of the variant model used by FeatureMapper is that it is not updated along with updates of the related feature model. Consequently, if the feature model evolves, the variant model has to be redefined from scratch.

Performance

When performing the transformation on the case study, Featuremapper took less than 1 second. The transformation is in fact almost instantaneous.

5.3.7 FeatureMapper with Pure::Variants

When applying the case study on featureMapper with Pure::variants, we first have to creat a new variant management project. Then, we create the Pure::Variants feature model, which is (partly) shown with Figure 21 (and was already created for the case study), and we import the target model of the case study. Thereafter, we have to create the mapping file. It consists of a .mapping.ccfm file, which is thus different from the one used with featuremapper only. Also, here we only have to specify the feature model, as we don't have to specify the target model manually. In fact, we can use several target models and the target models used are referenced automatically when we create a rule that maps one of their elements. To create these rules, we have to use the Pure::variants mapping view, which shows the different rules of the mapping, and the editor of the target model. Then, we just have to create a rule in the mapping view (right click-> new rule, then an editor appears, in which we have to enter the name of the feature, or create a feature expression) and then select model elements from the model and right click on the created rule.

After having created the mapping, we create a new variant with Pure::Variants (we can also create it just after having created the feature model), which as to be linked with the

feature model. Finally, we can transform the base model by clicking on “Transform” in the Pure::Variants mapping view and select the variant model to use.

Expressiveness

We were able to express every concept of the case study, but of course because the case study was implemented with the same tool. But, as explained in section 4, the expressiveness of the Pure::variants tool used with FeatureMapper is fairly good. For example, with Featuremapper only we cannot create constraints in the feature model, but here we can, and with a wide range of possibilities (“Requires”, “Conflicts”, “Discourages”,… constraints and also AND,OR,XOR, NOT relations within the constraint, such as shown with Figure 21). Also, the mapping supports more types of feature expressions than with Featuremapper, which can only create AND feature expressions in the mapping. With Pure::variants, the rules are created within the rule editor that in fact allows to create sophisticated feature expressions. The only concept that we could not create with this approach, but which isn’t included in the case study, is a group of OR features that has special cardinalities.

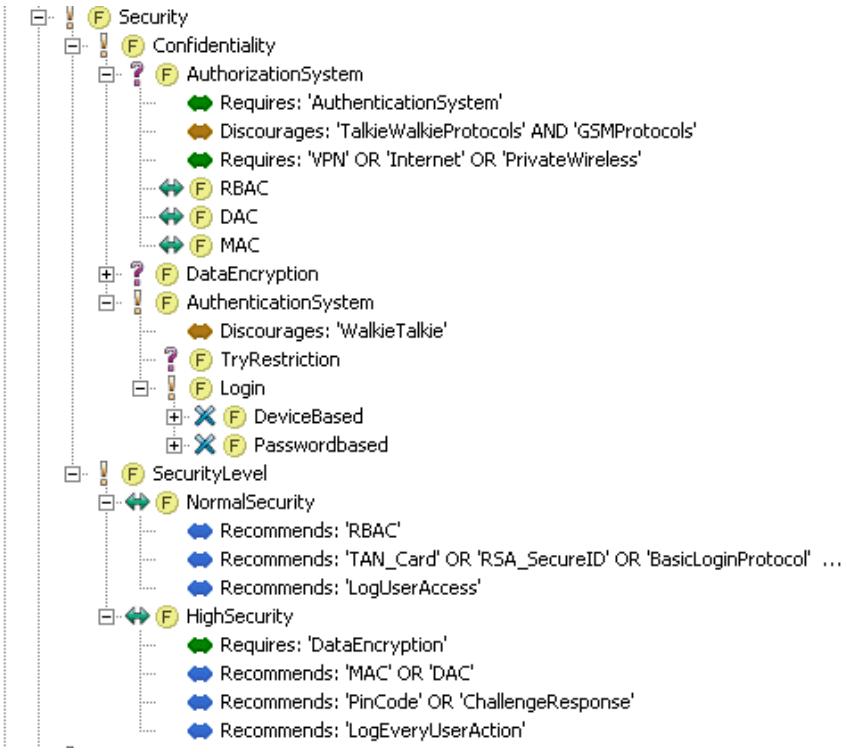


Figure 21: part of the Pure::Vraiants feature model for the REACT SPL

Usability

The steps to follow in order to define the mapping rule “observer” with Pure::Variants used with FeatureMapper are:

- Right click on the pure::Variants mapping view and then select “new rule”

- Create the rule in the rule editor that appeared by referring to the feature “Observer”, and click OK
- Select the model elements to map, i. e. the class “Observer”, its association link to the class “Mission” and its generalization link to the class “AidWorker”
- Add them to the rule via the right-click pop-up menu of the rule

So, creating the mapping rule “Observer” with this approach requires only 6 simple actions that are made in 7 clicks and typing the name of the rule (we can also select it in the selection dialog menu, adding a few more actions). Also, while typing the name of the rule, the editor supports auto-completion, which makes it easier and safer to use. It benefits from the same visualization functions than FeatureMapper, and the transformation is easy to process as well.

Unlike with FeatureMapper, the variants of the SPL are well handled with Pure::Variants. If the feature model evolves, the changes are automatically made in the variant models. The constraints between the features are also automatically checked and errors are signaled when they are violated, which was not always the case with FeatureMapper. Also, another usability advantage of Pure::variants is that the representation of the type of feature is made using icons, which are easily distinguished, compared to the cardinalities used by FeatureMapper. The Pure::Variants mapping view is also more usable the the (featuremapper) mapping, view, because it represents rules whereas FeatureMapper represents the feature diagram in its mapping view, from which the user has to select features to create rules. It is then harder to visualize the different created rules and also to create the rule with FeatureMapper, because we have to select the feature from a large feature model instead of writing its name using the auto-completion function. Also, with this Pure::Variants mapping view, the user can check which rules are used by a variant.

However, a small usability problem of Pure::variants is that it offers a wide range of different constraints for relations between features. In fact, if the meaning of Requires or Conflicts constraints are clear, it is less the case for constraints such as “influences” and thus it can be confusing.

Also, this approach, such as XSLT with Pure::Variants, uses a Pure::variants feature model. This model implements the links(such as relations between features, or here the links to the target model elements in the mapping) with absolute path references. So, the approach does not support the exportation of the project well, and it is for example not very usable with a versioning tool.

Performance

The performance of FeatureMapper with Pure ::Variants is similar to the one of FeatureMapper only, as it also took less than 1 second to perform the transformation.

5.3.8 XSLT

XSLT with Pure::Variants works with the Pure::Variants feature model. Consequently, we were able to import the feature model of the case study directly after having created a new Pure::Variants project. Then, we created a new .xsl file that will contain the mapping. In this file , we first have to define the used namespaces and the output type. Then, we create a general template that will contain the mapping. So, in this template, we copied the XML source of the UML target model (without its header). Then, for each mapping rule, we had to spot the related model elements in the text file, and then wrap those elements with the two following lines of code:

<xsl:if test= pv:hasFeature (“FeatureID”)>
</xsl:if>

With “featureID” referring to the id of the feature to map or its unique name (which is easier). We can also create feature expressions. Then, after having wrapped all the model elements to map, the mapping is complete, and we have to create a variant model with Pure::variants, such as with Pure::Variants with featureMapper. Then, for the transformation, we have to specify the input file (our xsl file) and the ouput directory in the properties of the configuration, space select one configuration and click on the transform button. After the transformation, we have to change the output file type from a XML file to a UML file, and then initialize the .umlclass diagram from the .uml file.

Expressiveness

Like Pure::Variants with FeatureMapper, XSLT with Pure::Variants uses the Pure::Variants feature model, which is also the modeling language used for the case study. Consequently, we were able to re-use the feature model of the case study directly. Also, the approach creates the mapping directly on the XML source code of the model, so we were able to re-use the target model from the case study as well. AS far as the mapping is concerned, we were also able to express all the mapping rules from the case study. For example, Figure 22 shows the mapping rule that maps the feature expression “InternalHumanresource” OR “ExternalResources” to the model element HumanResource. However, with XSLT, we couldn’t regroup multiple model elements in one “rule”, such as with FeatureMapper for example, because XSLT does not create rules but wrap the model elements with presence conditions. So, we had to keep the different model elements from a same rule separated, such as for the mapping rule Observer (see below).

```
      <type xmi:type="uml:PrimitiveType" href="pathmap://UML_METAMODELS/Ecore.metamodel.uml#EJavaObject"/>
    </ownedParameter>
  </ownedOperation>
  <ownedOperation xmi:id="_ckW38DWiEd-OG4fN-jL5IQ" name="hashCode">
    <ownedParameter xmi:id="_eLt3UDWiEd-OG4fN-jL5IQ" direction="return">
      <type xmi:type="uml:PrimitiveType" href="pathmap://UML_METAMODELS/Ecore.metamodel.uml#EInt"/>
    </ownedParameter>
  </ownedOperation>
  <ownedOperation xmi:id="_VDLwEDWjEd-OG4fN-jL5IQ" name="Human">
    <ownedParameter xmi:id="_WcUtQDWjEd-OG4fN-jL5IQ" direction="return"/>
  </ownedOperation>
</packagedElement>
 <xsl:if test="pv:hasFeature('ExternalResources') or pv:hasFeature('InternalHumanResource')">
<packagedElement xmi:type="uml:Class" xmi:id="_VVr2UDWYEd-OG4fN-jL5IQ" name="HumanResource">
  <generalization xmi:id="_OIsWwDwKEd-VwaPrEsL1Qw" general="_dG168DWXEd-OG4fN-jL5IQ"/>
  <ownedAttribute xmi:id="_WJNWQDWeEd-OG4fN-jL5IQ" name="STATUS_UNKNOWN" isStatic="true" isReadOnly="true" aggregati
    <type xmi:type="uml:PrimitiveType" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
  </ownedAttribute>
  <ownedAttribute xmi:id="_ORgvUDWgEd-OG4fN-jL5IQ" name="STATUS_NOTIFIED" isStatic="true" isReadOnly="true" aggregat
    <type xmi:type="uml:PrimitiveType" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
  </ownedAttribute>
  <ownedAttribute xmi:id="_32CLUDWgEd-OG4fN-jL5IQ" name="STATUS_UNAVAILABLE" isStatic="true" isReadOnly="true" aggre
```

Figure 22 : Part of the REACT mapping made with XSLT

Usability

For implementing the “Observer” mapping rule with XSLT, we have to do the following steps:

- Find the “Observer” class in the xsl file
- Wrap the class with <xsl:if test= ‘pv:hasFeature(Observer)’> and </xsl>
- Find the “CreatedBy” association in the file
- Wrap the association with <xsl:if test= ‘pv:hasFeature(Observer)’> and </xsl>

Creating this mapping rule with XSLT requires four actions only. Also, it doesn’t require more actions for the generalization link of Observer since it is contained by the Observer class in the XML source code. However, those actions are not simple to do, since we have to handle a very large file. For example, in our case, this file contains approximately 2000 lines, so we need to use a find function, which is not offered by the Eclipse text editor. We thus had to use another text editor to do so. Also, the XML code </xsl> must be placed correctly, and if we place it incorrectly or if we forget it, Eclipse just throws a “Impossible to parse .xsl file” which is not very helpful. Therefore, we have to pay attention more than with other approaches while creating the mapping with XSLT. Furthermore, while editing the file, we do not have any checking of the correctness of our mapping. So, for example, it is possible to make an error in the name of a feature, or to write a feature expression incorrectly, without any warning. Also, we cannot benefit from a mean of selecting the features for a feature expression.

The big drawback of this technique is that we have to handle the textual definition of the target model. All the mapping is made within this XML source code, and thus it is harder to create the mapping than with a graphical editor and also it is impossible to visualize the mapping, or the derived model from a variant before the transformation, such as with other methods. The XSL file is very large since it contains the source code of the UML target model. After the transformation, we also have to change the output file type from a XML file to a UML file, and then initialize the .umlclass diagram from the .uml file.

Such as shown with the creation of the Observer rule, we have to create the mapping of the association links separately from the mapping of the class. Therefore, if the mapping has to be changed, we have to modify the xsl file in two different locations.

Also, this technique requires knowledge in XML, XSL and XPATH in order to edit the XSL file and to create the XPATH expressions, which allow creating feature expressions.

This approach, such as Pure::Variants with FeatureMapper, uses a Pure::variants feature model. This model implements the links (such as relations between features) with absolute path references. So, the approach does not support the exportation of the project well, and it is for example not very usable with a versioning tool.

Performance

Performing the transformation with XLST took less than 5 seconds.

5.3.9 CVL

In order to use CVL for the case study, we first have to create a new "CVL visitor diagram". A CVL model is automatically created alongside containing the contents of the model, while the visitor diagram is used to visualize this model (otherwise, we can still visualize the CVL model as a tree). Then, we have to load the base model by clicking on the root of the CVL model and select "Load Resource". We also have to name the first element of the CVL model and set his base model property to the root element of the base model, which is in our case the package uni.lu. Thereafter, we can edit the CVL model by adding composite variability elements, OR choices,… . In fact, in the CVL approach, we use this CVL model to express the variability instead of a feature model. So, what we did is that we took the feature model of REACT along with us and for each feature that was relevant for the mapping (i.e. mapped to solution domain models elements) we created a "composite variability element". Then, to represent the mapping, we defined fragment substitutions for each composite variability elements. Since the case study was created using negative variability techniques, we also used negative variability with CVL. Consequently, each fragment substitution refers to a placement fragment, but no replacement fragment, because we want to remove the model elements and not replace them with other elements. The placement fragments are made by selecting elements in the base model, and then clicking on "create placement fragments from selection and contained elements" from the right-click pop-up menu of a composite variability element. This allows us to map the class and its contained elements such as methods and attributes in one action.

After having defined all the fragments substitution, the mapping is complete. Then, we have to create a variant. With, CVL creating a variant is made by selecting the composite variability elements from the CVL visitor diagram view and then clicking on the "create variant" button, which creates the variant in the CVL model. Finally, we have to click on the transform button to process the transformation, which creates one product model for each variant defined in the CVL model.

However, this approach led us to a problem: since we could not map fragment substitutions to the non-selection of a composite variability element, but only to the selection

of one element, using “standard” variants would led to remove from the model the elements that we would like to keep, so we created variants with the meaning that selected composite variability elements are the one that we want to remove. We also had to change the cardinalities of our OR decompositions in order to make it possible to select no elements, which means selecting every “features” of the decomposition.

Expressiveness

The difference between CVL and other methods is that it uses a CVL model made of composite variability elements rather than a feature model to express the variability of the product line. Yet, we could translate most of different concepts of the REACT feature model in the CVL model. In fact, we could express mandatory features, optional features, OR groups and AND groups. A mandatory feature is here a simple composite variability elements and the AND group is represented as composite variability children. Optional features are represented as a single child from an OR group, and OR groups are defined with an OR decomposition that has min and max cardinalities (such as in Figure 21) .However we were not able to create constraints between the composite variability elements (CVL allows creating those kind of constraints, but they are not checked when creating variants so, apart from informing the user, they are useless).

For the target model we have been able to import the model of the case study without problem.

The mapping was, as we discussed above, not able to express actions related to the non-selection of a feature. The fragment substitutions must in fact be linked to a composite variability element, which cannot be negated. If a fragment substitution can be reused and linked to multiple model elements, making it like an OR feature expression, we cannot express AND feature expressions with CVL. In our application of CVL to the case study, we inverted the meaning of the selection of features. Therefore, if two composite variability elements are linked to a removal fragment substitution, it will be removed if at least one of the composite variability is selected, and kept if none are selected which means that both “features” would be selected. So, in our case, we can express AND feature expressions but not OR feature expressions. So, for the mapping rule of our case study “ InternalHumanResource OR ExternalHumanResource”, which states that if one of these features are selected, the mapped model elements must not be removed, we had to modify the CVL model. In fact, we added a third sibling composite variability element to the elements InternalHumanresource and ExternalResource that we named InternalORExternal, and to which we linked the fragment substitution that removes the model elements. In that way, if this element is selected, it means that the original feature rule is not satisfied (since we had to invert the meaning of the selection) and then the elements will be removed. Also, we couldn’t use constraints on this element that would say that it is selected only if maximum one of the two other elements are selected, which would have ensured that it is well a OR feature expression.

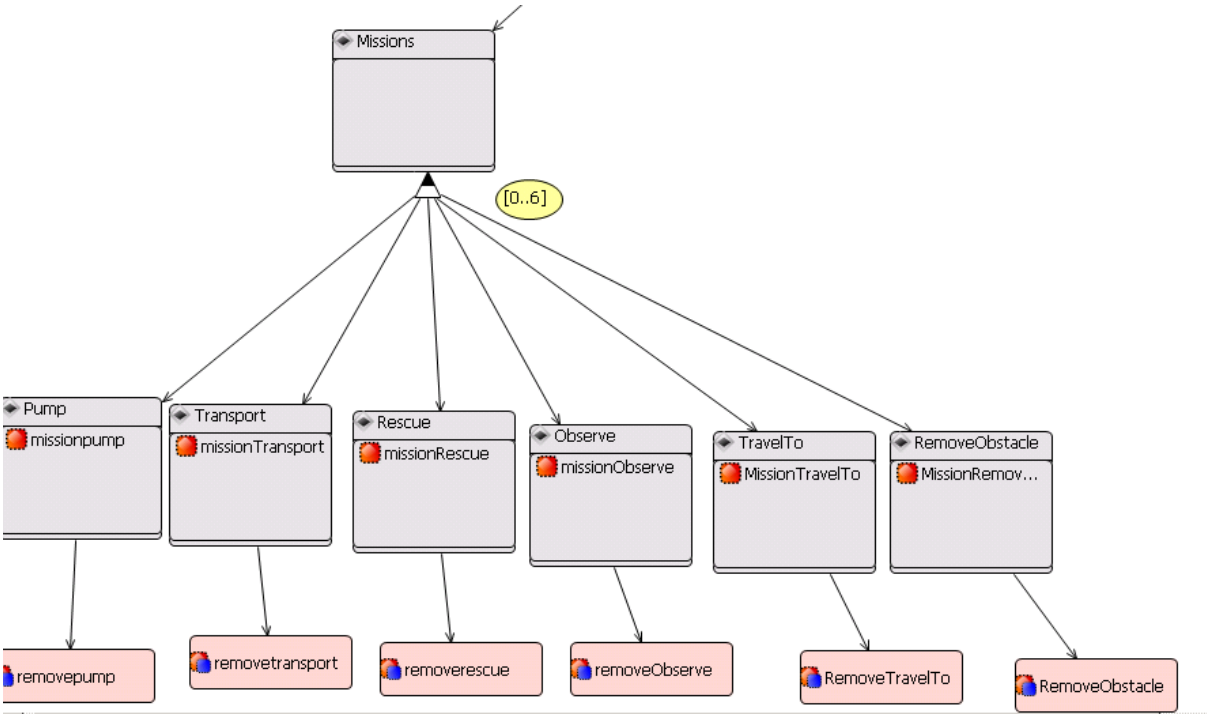


Figure 23 : Part of the CVL model for the REACT SPL

Usability

Assuming that we already created the composite variability elements of the CVL model, the steps to follow in order to create the Observer mapping rule are:

- Select the class “Observer” from the target model
- Add the association link “CreatedBy” to the selection
- Right-click on the “Observer” composite variability element from the CVL model and select “Create placement fragment from selection and contained objects”
- Name the placement as “Observer”
- Create a fragment substitution in the CVL visitor diagram and name it “RemoveObserver”
- Link the fragment substitution to the composite variability element “Observer”
- In the fragment substitution properties view, select “Observer” as placement fragment.

The realization of the mapping rule is here a bit more complex than with the other tools. It requires 7 actions that are made in 13 clicks and also requires naming 2 newly created elements. It requires dealing with some graphical editing too, as we create the fragment substitution and link it to a composite variability element. Moreover, in our case we didn’t have to specify replacement fragments, and replacement bindings, but they are required when using positive variability, and thus add more required actions in that case.

As we saw in the previous sections, using negative variability with CVL is possible. However it brings usability problems since we cannot link fragment substitutions to the non-selection of an element. We then have to invert the meaning of the selection of composite variability elements, which can be confusing and also requires a few changes in the CVL model, such as for the cardinalities of the decompositions. This is avoided only when the removal of elements is caused by the presence of a feature in the variant model, and not by its absence, but in general that is not the case. So, we find that CVL is not very usable for the negative variability approach and was more developed as a positive variability technique.

A big usability drawback of CVL is that it doesn't use a feature model. As we saw in [Fle09], they still prefer to use a feature model to have a view of the variability of the product line, and we also did that when applying the case study on CVL. In fact, the CVL model doesn't express the variability as well as a feature model, partly because it also contains mapping elements. For example, for our case study, there were a majority of 1-1 mappings, and so the CVL model looks like the class diagram with a hierarchical organization. Also the CVL only models the "features" where variation occurs, which is not the case of a feature model, which contains mandatory features for example. The CVL model cannot express constraints too, which brings usability problems when we have to create variants, but also when we have to model the variability. For example, a constraint such as "MissionObserve" requires "Observer" cannot be expressed in CVL and so we cannot ensure that created variants do not violate this kind of constraint.

Also, as we want to express OR feature expressions with negative variability techniques or AND features expressions with positive variability techniques, we have to create more composite variability elements that represent those expressions. It thus requires more effort than with other approaches and it is also less secure because we cannot express the constraints that would validate the expression, since the constraints are not checked while creating and using the variants.

With CVL, we have to specify the root element of the base model. So, if we can work with many library models (models that contains the elements to add in the case of positive variability), we can only work with one base model at a time. Moreover, because the approach places the mapping and the description of the variability onto the same model, working with two base models would here require to re-implement both of the mapping and the variability description, whereas VML* has the same problem (see further) but only has to re-implement the mapping.

Also, when specifying properties such as the root element of the base model, we have to select it from a list. However, this list can be quite long. For example, for the root element of the base model, we have a list of all the elements of all the loaded models.

The variant models of CVL also bring a few usability problems. First, when we want to process the transformation, all the created variants are used, and for each one a product model is generated, which only differs by a number in its name (and of course its content). It would be more usable to allow the user to select which variant he wants to select. Also, variants are not modified along with the evolution of the CVL model, and they also cannot be modified. Finally, they do not provide constraint checking for the selection of composite variability elements, so it is possible to create a conflicting variant.

With CVL, we cannot have a direct view of what would be derived from a variant. Also, the target models have to be handled with a XML file viewer, unless we would had implemented CVL API's for benefiting from a graphical editor. Otherwise, CVL offers a way of visualizing the elements that are contained in a placement or replacement fragment, such as shown with the figure below.

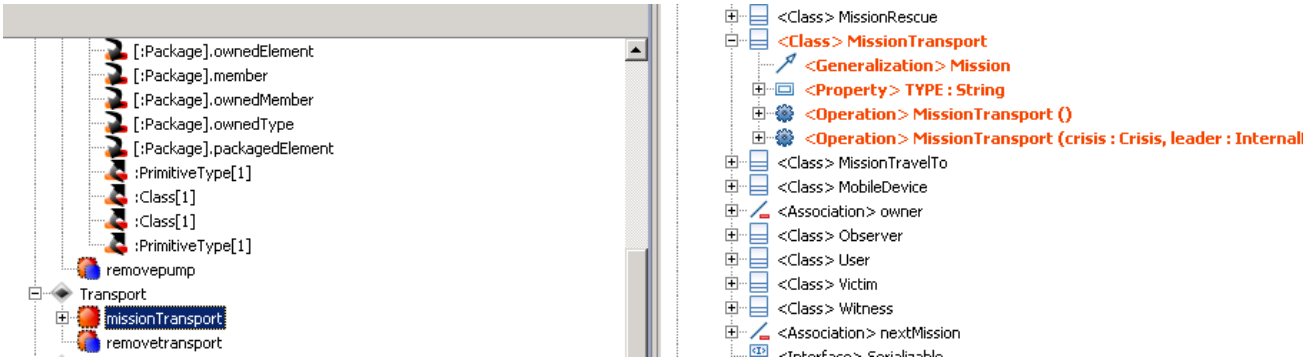


Figure 24 : Visualisation of the target model elements linked to a placement fragment with CVL

Finally, an advantage of CVL compared to other approaches is that it can re-use mapping elements such as placement fragments or fragment substitutions. We didn't have to use that in our case study but this could be useful for a bigger SPL, where different mapping rules would requires the same actions (or a part of those actions).

Performance

Performing the transformation with CVL took less than one second.

5.3.10 VML*

For the application of VML* onto the case study, we reused the VML*language VML4ARCH, which focuses on mappings for architectural models. So, the first step to do was to create a new VML4ARCH project. Since VML4ARCH works with feature diagram developed in FMP only, we created a new feature model with FMP and copied the features from the feature model case study. We also imported the UML diagram from the case study. Then, we edited the vml4ARCH file contained in the VML4ARCH project by first specifying the path of the input target model and the input feature model and then specifying each mapping. For each mapping, we re-used the function "remove ("element")" from the VML4ARCH specification. Thereafter, we created a new variant of the feature model, which is created with FMP in the same model as the feature diagram, by right-clicking on the root element of the feature diagram. Finally, in order to process the transformation, we had to compile the VML4ARCH file and then click on "configure" and select the feature model. So, the transformation creates product models for each configuration defined in the feature model.

Expressiveness

VML4ARCH uses features models made in FMP. With these models, we were able to express mandatory and optional features, as well as AND, OR, XOR decompositions and features cardinalities. In fact, each feature or feature group defined in FMP possesses minimum and maximum cardinalities (see Figure 25), which are used to define the previously listed concepts. However, we were not able to express constraints between features.

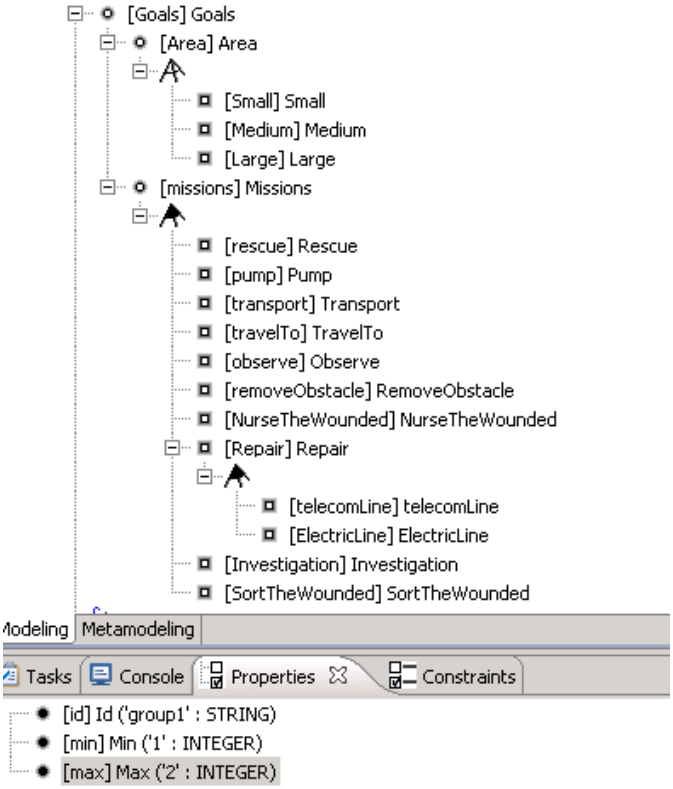


Figure 25 :Part of the FMP feature model of the REACT SPL

For the target model we were able to express every of its concepts.

For the mapping model, we were able to express 1-1 mapping rules as well as feature expressions in a rule, such as shown with the rule "variant for not (or(InternalHumanResource, ExternalResources)) " in Figure 26 . We also see that we can also create rules for the non-selection of a feature with the "not" operator. We re-used the VML4ARCH action remove (element) in order to remove the model elements specified by the rules. Also, compared to VML4RE, it is not possible to express tracing links while using VML4ARCH (but they are not required in our case study).

```
variant for not (Transport){
remove ("Transport");
}
variant for not (RemoveObstacle){
remove ("RemoveObstacle");
}
variant for not (or (InternalHumanResource,ExternalResources)){
remove ("HumanResource");
}
variant for not (InternalHumanResource){
remove ("InternalHumanResource");
}
variant for not (ExternalResources){
remove ("ExternalResources");
}
variant for not (MedicalServices){
remove ("MedicalServices");
}
variant for not (Observer) {
remove ("Observer");
remove ("CreatedBy");
```

Figure 26 : Part of the VML mapping model of the REACT SPL

Usability

With VML*, the creation of the mapping rule “Observer” requires the following steps:

- write the rule “variant for not (“Observer”){} in the VML4ARCH model
- wite the action “remove (“Observer”)” between the brackets of the rule
- Write the action “remove (“CreatedBy”)” after the previous action

So, this mapping rule is quite easy to create with VML4ARCH. It does not require special knowledge like XSLT and AHEAD do. It is however a bit less easy than with the approaches like FeatureMapper that comes with a graphical editor, because here we have to ensure that we are referencing the good features. For example, it would be possible to write “Observe” instead than “Observer” without being warned since the model element Observe also exists. That problem is less likely to occur with those other approaches, because the user has to select the model elements directly from the model, and also because it can see which elements are mapped to which rules thanks to the visualization functions. VML* does not bring those kind of visualization functions. Consequently, it is impossible to view the effects of a mapping onto a target model.

Such as the approach of XSLT with Pure::Variants and the approach of AHEAD, VML* requires defining the mapping in a text editor rather than in a graphical editor, which is less user-friendly. But, unlike the two other approaches, VML* brings a few user-friendly functionalities. In fact, the VML text editor supports the coloring of reserved words (such as “remove” or “variant”), and also it checks if the referenced paths and features exists. If they do not exist, it shows a warning to the user. Therefore, creating a mapping is less prone to writing errors with VML* than with XSLT or AHEAD.

Also, in our application of VML* to the case study, we re-used the concepts of VML4ARCH. If VML4ARCH brought what we needed for our case study, we were also limited by it. In fact, we only could use the remove action and thus negative variability. Also, it would had been possible to develop a VML language dedicated to class diagrams, which would for example offer an action that removes the associations between classes when one of their target is removed.

With FMP, the variant configurations are contained within the feature model. This makes the approach less usable since it is not possible to specify which configuration we want to use for the transformation. In fact, the transformation will be processed for every configuration contained in the specified feature model, such as with CVL.

Finally, with VML4ARCH, we have to specify the target model at the beginning of the VML4ARCH model. Consequently, it is not possible to use more than one target model at a time.

Performance

Performing the transformation with VML4ARCH requires compiling the VML4ARCH (if something has been modified) and then we can process the "configure" action. Both actions took about 5 seconds during our tests, so the total time needed to perform the transformation is 10 seconds.

5.4 Conclusion of the empirical comparison

In the last section we compared the different tools by applying the criteria that we had defined before onto them. Now, we will resume the main facts that we discovered during this comparison, for the expressiveness, usability and performance of those tools. The table below shows our evaluation of the different tools for each criterion.

	Usability	Expressiveness	Performance
FlocosGPL	---	--	/
AHEAD	--	--	+-10 seconds
FeatureMapper	+	+	<1 second
Pure ::Variants with FeatureMapper	++	++	<1 second
XSLT	-	+	<5 seconds
CVL	+-	+-	<1 second
VML*	+	+	+-10 seconds

Table 5 : Conclusion of the the empirical comparison

The comparison of the usability of the tools based on our experiences showed that there were more usable tools than others. First, FLOCOSGPL does not bring automatic transformation abilities, and so is clearly the less usable. The approach of AHEAD also has a poor usability. It does not bring any visualization technique and requires handling low-level XML files, a lot of different files for the refinements and also no mean of checking constraints. Furthermore, deriving a model requires writing a complex command line. XSLT

also handles low-level XML and does not bring visualization functions for the mapping. However, it is more usable than AHEAD because it does not separate the mapping into different files, and also because it uses the Pure::Variants feature models and variant models, which makes it easier to create a valid variant. The other approaches do not require handling low-level XML, which makes them easier to use. However, there still were some usability problems. In the case of CVL, the mapping in itself is quite complicated to create (with the definition of refinements,..) and it cannot handle negative variability well. Its variants are not checked about their validity as well, and it is not able to show the effects of the mapping for a variant . With VML4ARCH, the mapping was easier to create, but we still couldn't benefit from visualization functions. That was possible with FeatureMapper and FeatureMapper with Pure::Variants. However, with FeatureMapper, the feature model used brought some usability issues, and also the variant checking did not work for every case.

There were also some differences with the expressiveness of the different approaches. The approaches of FLOCOSGPL and AHEAD do no express the features of the product line, and their mapping is also limited. In fact, FLOCOSGPL is not able to express feature expressions for the mapping, and in AHEAD it is possible but they cannot be linked to their features, since they are not modeled. The approach of CVL does not use a feature model as well, but it relies on the CVL model in order to express the variability of the product line, which showed to be as expressive as the feature model of VML and FeatureMapper, whre we could express anything from the case study, excepted from the relations between the features, which we could express only with the Pure::Variants feature model. We encountered another problem with the expressiveness of CVL. In fact, it was not able to express the fact that an action of a mapping rule could be linked to the non selection of a feature (in CVL, a "composite variability element"). With XSLT, we had the problem that we could not put the actions in rules, but that the actions were directly expressed on the model itself. So, we had to decompose some of the mapping rules. With FeatureMapper, we also had a few expressiveness problems, because (along with the fact that constraints couldn't be expressed) we could not expressed OR feature expressions for the mapping.

The comparison of the performance of the different tools shows that there is little difference for that point of view. In fact, FeatureMapper, Pure::Variants with FeatureMapper and CVL took less than one second to perform the transformation. It took us a little bit more time to perform the transformation with XSLT, but it was still under 5 seconds. Finally, VML* and AHEAD were a bit less efficient with a time of about ten seconds. But, for AHEAD, we also have to take into account the time to write the full command line, which is more than ten seconds.

6 Conclusion

In this master's thesis, we focused on different approaches that bring a solution to the variability mapping problem (i.e. the problem of specifying links between the features of a product line and elements of the product line architectural models), which can be used along with model driven engineering techniques to automate the product derivation process. If used in an efficient manner, these techniques can ease the transformation process, as well as decrease the efforts needed in time and costs and improve the quality of the produced software.

To analyse these different approaches, we first presented the different concepts that belong to the context of the problem. Thereafter, we presented the different variability mapping approaches that we analysed in this work. Then, we presented a literature-based comparison of them, which was followed by a more empirical comparison based on the application of a case study consisting of implementing the REACT product line with each tool.

More than a state of the art, our main contributions are that we brought an overview of what the most-known existing approaches of the variability problem can do to handle it and automate the product derivation which is key to the product line paradigm. We demonstrated their advantages and disadvantages in two ways: firstly, we based ourselves on their documentation to point out their differences about their (explicit) expressiveness, usability, adaptability and maturity. Secondly we showed with a case study how the different tools handle the problem in practice, and we analysed their expressiveness for this case, their usability and their performance.

Our work presents some limits: as we discussed earlier, the domain of software product lines is still moving rapidly. So, the approaches that we analysed can sill evolve, and new approaches to the problem, or new implementations, can also be found or developed. Moreover, some approaches that we studied also were still in development, and so their characteristics can be modified. Also, for both comparisons, we only analysed what was relevant for our comparison criteria. These criteria do not cover every possible aspect of such comparisons, and so they can still be refined, and it would also be possible to add new criteria, even if we think that our criteria already cover a good range of aspects. Finally, for the literature-based comparison of the approaches, we based ourselves on the documentation of the tools. However, this documentation is not always complete, and so we had to remove some approaches from the comparison, or base ourselves on limited documentation to compare them. The same can be said about the empirical comparison, where a few tools were not usable anymore due to compatibility problems with actual frameworks, or were even not available.

About the future works and perspectives that our work involves, we could focus on the actual limits of the work. For example, we could refine our comparison criteria. An interesting criterion would be to analyse the formal semantics of the feature modeling language of each approach, or of its mapping language. Also, we deliberately chose to limit the scope of our research, and so we did not analyse all the possible approaches to the problem. We could therefore extend the comparison to other approaches, such as in the domain of decision-

oriented approaches, where the comparison found in [SRG11] could be extended to more approaches. Another future work could be to use the tools on a “real life product line” that is bigger and more complex than the one we used (i.e. the REACT product line, which is yet already bigger than the product lines used as example in the documentation of the tools we used). In a larger scale, the domain of product line engineering also offers some perspectives. It is viewed as the emerging paradigm that responds to the need of reducing time and costs of software development. Our comparisons of existing tools showed that they are not really mature yet and that their support to the automated product derivation process needs further improvements.

7 Bibliographie

[ABFG00] M. Anastatsopoulos, J. Bayer, O. Flege, C. Gacek. (2000) A Process for Product Line Architecture Creation and Evaluation PuLSE-DSSA – Version 2.0, Fraunhofer.

[Agi11] Agile modeling. Agile Modeling. [Online]. http://www.agilemodeling.com/artifacts/classDiagram.htm

[AHE11] The AHEAD Tool Suite (ATS). [Online]. http://www.cs.utexas.edu/users/schwartz/ATS.html

[Alf09] Mauricio Alférez, João Santos, Ana Moreira, Alessandro Garcia2, Uirá Kulesza, João Araújo, Vasco Amaral. (2009) Multi-View Composition Language for Software Product Line requirements. in Proceedings of the 2nd Int. Conference on Software Language Engineering (SLE), Denver, USA.

[Amp] Ample project. VML4RE user manual.

[Anq08] Birgit Grammel, Ismênia Galvão, Joost Noppen, Safoora Shakil Khan, Hugo Arboleda, Awais Rashid, Alessandro Garcia Nicolas Anquetil. (2008) Traceability for Model Driven, Software Product Line Engineering.

[Bal08] Florencia Balbastro. (July 2008) 081010_REQS_BECS.

[BeDa06] D. Beuche M. Dalgarno. (2006) Software Product Line Engineering with Feature Models. Overload Journal, 78:5–8.

[Big11] BigLever Software Inc. (2011, June) BigLever Software Gears. [Online]. http://www.biglever.com/solution/product.html

[BLT08] G. Botterweck, K. Lee, S.Thiel. (2008) Automating Product Derivation in Software Product Line Engineering. In Software Engineering, pp. 177–182.

[BSR04] Don Batory, Jacob Neal Sarvela, Axel Rauschmayer. (2004) Scaling Step-Wise Refinement; Proceedings of the 25th International Conference on Software Engineering, Prtland, Oregon.

[BuDo09] Thomas Buchmann, Alexander Dotor. (2009) Constraints for a fine-grained mapping of feature. In 1st International Workshop on Model-Driven Product Line Engineering (MDPLE 2009) University of Twente, The Netherlands, CTIT Workshop Proceedings, June 2009, p.9-17.

[BuDo091] Alexander Dotor Thomas Buchmann. (2009) Towards a Model-Driven Product Line for SCM Systems. in: Proceedings of the 13th International Software Product Line Conference (SPLC), Vol. 2, San Francisco, CA, August 24-28, 2009, p. 174-181.

[BuDo092] Thomas Buchmann, Alexander Dotor. (2009) Mapping Features to Domain Models in Fujaba. in: Proceedings of the 7th International Fujaba Days: Pieter van

Gorp (Ed.), Eindhoven, The Netherlands, November 16-17, 2009, p. 20-24.

[CHE04] Krzysztof Czarnecki, Simon Helsen, Ulrich Eisenecker. (2004) Staged Configuration Using Feature Models. Third Software Product-Line Conference (SPLC'04), 2004, vol. 3154, Boston, USA, Springer-Verlag, pp. 266-283.

[CHE05] Krzysztof Czarnecki, Simon Helsen, Ulrich Eisenecker. (2005) Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice, vol. 10, issue 2, pp. 143 - 169.

[ClNo01] P. Clements, L. Northrop. (2001) Software Product Lines: Practices and Patterns. Addison-Wesley.

[ClNo03] P. Clements, L. Northrop. (2003) A Framework for Software Product Line Practice. http://www.sei.cmu.edu/plp/framework.html: Software Engineering Institute.

[CVL] CVL 1.2 user guide.

[Cza05] Krzysztof Czarnecki, Michał Antkiewicz,Chang Hwan Peter Kim, Sean Lau, Krzysztof Pietroszek. (2005) fmp and fmp2rsm: Eclipse Plug-Ins for Modeling Features using model templates. In Proceeding OOPSLA '05 Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.

[CzAn05] Krzysztof Czarnecki, Michael Antkiewicz. (2005) Mapping Features to Models: a Template Approach based On Superimposed Variants. ACM SIGSOFT/SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'05), vol. 3676, Tallinn, Estonia, Springer-Verlag, pp. 422 - 437.

[CzHe03] K. Czarnecki, S. Helsen. (2003) Classification of Model Transformation Approaches. In 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA.

[DSB04] Sybren Deelstra, Marco Sinnema, Jan Bosch. (2004) Experiences in Software Product Families: Problems and Issues during Product Derivation. In SPLC3, pages.

[DSB041] Sybren Deelstra, Marco Sinnema , Jan Bosch. (2004) Product derivation in software product families: a case study. J. Syst. Softw., 74(2):173–194.

[EBB05] M. Eriksonn, J. Borstler, K. Borg. (2005) The pluss approach - domain modeling with features, use cases and use case realizations. in: Proc. of SPLC 2005.

[Fea11] FeatureMapper. FeatureMapper - Mapping features to models. [Online]. http://featuremapper.org/

[Fin94] Finkelstein, O. C. Z., Gotel A. C. W. (1994) An analysis of the requirements traceability problem. In Proceedings of the First International Conference on requirements Engineering, pp. 94-101.

[Fle09] Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, Gøran K. Olsen, Andreas Svendsen, Xiaorui Zhang. (2009) A Generic Language and Tool for Variability Modeling. SINTEF Report.

[fmp11] Generative Software Development Lab. [Online]. http://gsd.uwaterloo.ca/fmp2rsm

[FNS09] Carlos Nebrera, and Pablo Sanchez Lidia Fuentes. (2009) Feature-Oriented Model-Driven Software Product Lines: The TENTE approach. Proc. of the Forum of the 21st International Conference on Advanced Information Systems (CAiSE), Eric Yu, Johann Eder and Colette Rolland (Eds), CEURS Workshops(453):67-72, Amsterda.

[FUJ11] Fujaba Tool Suite. [Online]. http://www.fujaba.de/no_cache/home.html

[GFdA98] M. L. Griss, J. Favaro, M. d' Alessandro. (1998) Integrating Feature Modeling with RSEB. Proceedings of the Fifth International Conference on Software Reuse (ICSR), IEEE Computer Society Press, Los Alamitos, CA, pp 76-85.

[GNFL09] Bruno Gadelha, Ingrid Nunes, Hugo Fuks, Carlos de Lucena. (2009) An approach for developing Groupware Product Lines based on the 3C Collaboration Model. In CRIWG: International Workshop on Groupware Conference, pp. 328-343.

[GrVo09] I. Groher, M. Voelter. (2009) Aspect-Oriented Model-Driven Software Product Line Engineering. In Transactions on Aspect-Oriented Software Development VI: Special Issue on Aspects and Model-Driven Engineering, Springer-Verlag, Berlin, Heidelberg.

[Hau04] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Arnor Solberg. (2004) An MDA®-based framework for model-driven product derivation. In SEA, pages 709–714. ACTA Press.

[Hei09] Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza, Ana Moreira, Awais Rachid. (2009)] Relating Feature Models to Other Models of a Software Product Line: A Comparative Study of FeatureMapper and VML*. In Proc. of the 2nd Int. Conference on Software Language Engineering (SLE) pages 82-102.

[HeTr03] P. Heymans, J.-C. Trigaux. (2003) Software Product Lines: State of the art. Technical Report EPH3310300R0462/215315, PLENTY project.

[HJSW08] F. Heidenreich, J. Johannes, M. Seifert, C. Wende. (2008) Closing the gap between modelling and Java. In Proceedings of the 2nd International Conference on Software Language Engineering (SLE 2009). Denver, Colorado, USA. Springer, LNCS 5969.

[IEE00] IEEE Standard. (2000) Recommended Practice for Architectural Description of Software-Intensive Systems.

[IKJ10] P. Istoan, J. Klein, J.-M. Jezequel. (2010) Survey and Classification of Software Product Line Variability Modelling techniques.

[Ist10] Paul Istoan, Jacques Klein, Jean-Marc Jezequel. (2010) Survey and Classification of Software Product Line Variability Modelling Techniques. Submited to IEEE Transactions on Software Engineering. Status: under review.

[Jac97] I. Jacobson,M. Griss, P. Jonsson. (1997) Software Reuse. Architecture, Process and Organization for Business Success. In Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering.

[Kan90] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson. (1990) Feature-Oriented Domain Analysis feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[Kan98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, Moonhang Huh. (1998) FORM: A feature-oriented reuse method with do-main-specific reference architectures. In annals of Software engineering, volume 5, number 1, pages 143-168, Springer-Verlag.

[KGM10] Jorg Kienzle, Nicolas Guelfi, Sadaf Mustafiz. (2010) Crisis Management Systems : A Case Study for Aspect-Oriented Modeling. Trans. Aspect-Oriented Software Development (TAOSD), 7:1-22.

[Kic97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes. (1997) Aspect Oriented Programming.In Proceedings of the European Conference on Object-Oriented Programming, vol.1241. pp. 220–242.

[KLD02] K. Kang, J. Lee, P. Donohoe. (2002) feature Oriented product Line Engineering. IEEE Software, vol. 19, no. 4, pp. 58-65.

[Krz02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, Ulrich W. Eisenecker. (2002) Generative Programming for Embedded Software: An Industrial Experience Report. First ACM SIGPLAN/SIGSOFT International Conference on Generative Programming and Component Engineering (GPCE'02), Lecture Notes in Computer Science, vol. 2487, Pittsburgh, USA.

[Las09] LASSY (Optimal Security). (2009) Software Product Line Analysis and Design of React: version 0.7.

[Lec11] M. Leclercq. (2011) Model driven development of crisis management applications in a software product line. http://wiki.lassy.uni.lu/Special:LassyBibDownload?id=3125.

[MaHe05] A. Maccari, A. Heie. (2005) Managing infinite variability in mobile terminal software. Softw., Pract. Exper. 35(6): 513-537.

[MeHe07] A. Metzger P. Heymans. (2007) Comparing Feature Diagram Examples Found in the Research Literature. Technical Report, TR-2007-01, Software Systems Engineering, University of Duisburg-Essen.

[MKR06] T.Mens, G. Kniesel, O. Runge. (2006) Transformation dependency analysis : A comparison of two approaches. Serie L'objet - logiciel, base de donnees, reseaux (2006).

[NES07] NESS Technologies and Systems Group. (2007) CMS : Crisis Management System.

[OMG05] Object management Group. (2005) OMG Unified Modeling Language (OMG UML), Superstructure.

[OMG11] OMG. OMGWiki. [Online]. http://www.omgwiki.org/variability/doku.php

[Par76] D.L. Parnas. (1976) On the Design and Development of Program Families. IEEE Transactions on Software Engineering, vol. 2, no. 1, pp. 1-9.

[PBvdL05] K. Pohl, G. Bockle, F. van der Linden. (2005) Software Product Line Engineering. Springer-verlag New-York Inc.

[Per07] G. Perrouin. (2007) Architecting software systems using model transformations and architectural framework. PhD thesis, FSTC, Université du Luxembourg, and Institut d'Informatique, Université de Namur, Sept 2007.

[PfPi08] M. Pfeiffer, J. Pichler. (2008) A Comparison of Tool Support for Textual Domain-Specific languages. In The 8th OOPSLA Workshop on Domain-Specific Modeling.

[PKGJ08] G. Perrouin, J. Klein, N. Guelfi, J.-M. Jezequel. (2008) Reconciling Automation and Flexibility in Product Derivation. In 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland.

[Pur] Pure-Systems GmbH. Pure:Variants User guide Version 3.0.

[Pur09] Pure Systems GmbH. (2009) pure:variants User's Guide version 3.0.

[Pur11] Pure Systems GmbH. Pure:Variants. [Online]. http://www.pure-systems.com/pure_variants.49.0.html

[RBSP02] Matthias Riebisch, Kai Bllert, Detlef Streitferdt, Ilka Philippow. (2002) Extending feature diagrams with uml multiplicities. 6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA.

[RoLa02] B. Robert, C. Lajtha. (2002) A new approach to crisis management. In Journal of Contingencies and Crisis Management, Volume 10, Issue 4, pages 181-191.

[Scha02] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer G. Kappel. (2002) A Survey on Aspect-Oriented Modeling approaches.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux. (2006) Feature Diagrams: A Survey and A Formal Semantics. in Proceedings of 14th IEEE International Requirements Engineering Conference (RE'06), pp. 139-148.

[SHTB07] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps. (2007) Generic semantics of feature diagrams. in Computer Networks, volume 51, issue 2, pp. 456-479.

[Spring] Spring. (2011, Aug.) Spring source Community. [Online]. http://www.springsource.org/

[SRG11] K. Schmid, R. Rabiser P. GrünBacher. (2011) A comparison of decision modeling approaches in product lines. In VaMoS 2011.

[SSU98] M. Seeger, T. L. Sellnow, R. Ulmer. (1998) Communication, organization and crisis. In M. E. Roloff (Ed.), Communication Yearbook 21. Thousand Oaks, CA: Sage.

[Sven10] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen Gøran K. Olsen. (2010) Developing a Software Product Line for Train Control: A Case Study of CVL. In Proceedings of the 14th International Software Product Line Conference, 13-17 September, Jeju Island, South Korea, pages 106--120.

[TBD06] Salvator Trujillo, Don Batory, Oscar Diaz. (2006) Feature Refactoring a Multi-Representation Program into a Product Line. GPCE 2006. In 5th International Conference on Generative Programming and Component Engineering. Portland, Oregon, USA..

[TBD07] Salvator Trujillo, Don Batory, Oscar Diaz. (2007) Feature Oriented Model Driven Development: A Case Study for Portlets, on 29th International Conference on Software Engineering (ICSE'07).

[vdL02] F. van der Linden. (2002) Software Product Families in Europe: The Esaps & Café Projects. IEEE Software, v.19 n.4, p.41-49.

[vdML04] T. von der Massen, H. Lichter. (2004) Deficiencies in Features Models. In Proceedings of SPLC'04 Workshop on Software Variability Management for Product Derivation - Towards Tool Support.

[vGBS01] Jilles van Gurp, Jan Bosch, Mikael Svahnberg. (2001) On the Notion of Variability in Software Product Lines. In Proceedings of the 2nd Working IEEE / IFIP Conference on Software Architecture (WICSA), pages 45–54. IEEE Computer Society.

[Whi96] J. Whitey. (1996) Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR010, Software Engineering Institute, Carnegie Mellon University.

[XAK11] The AHEAD Tool Suite. [Online]. http://www.cs.utexas.edu/users/schwartz/ATS/fopdocs/xak.html

[XSL11] W3C. [Online]. http://www.w3.org/TR/xslt

[ZiJe06] T. Ziadi J.-M. Jézéquel. (2006) Software Product Line Engineering with the UML: Deriving Products. Chapter in Software Product Lines: Reasearch Issues in Engineering and Management, Käkölä Timo , López Juan Carlos Dueñas, pp. 557-596, (Springer-Verlag).

[ZiJe061] T. Ziadi, J.-M. Jézéquel. (2006) Software Product Line Engineering with the UML : Deriving products. chapter in Software Product Lines: Reasearch Issues in Engineering and Management, Käkölä Timo , López Juan Carlos Dueñas, pp. 557-596, (Springer-Verlag), (ISBN : 3-540-33252-9).

[ZJH02] T. Ziadi, Loïc Hélouët, J.-M. Jézéquel. (2002) Modeling behaviours in Product Lines. In Proceedings of International Workshop on Requirements Engineering for Product Lines (REPL02) co-located with RE 02, 2002, Essen/Germany, Technical report AVAYA labs ALR-2002-033, pp 33-38.

[ZJH03] T. Ziadi, L. Hélouët, J.-M. Jézéquel. (2003) Modélisation de Lignes de Produits en UML. In Proceedings of Langages et Modèles à Objets (LMO03) Vannes/France, 2003. Publié dans la revue l'Objet volume 9 – n1-2/2003.

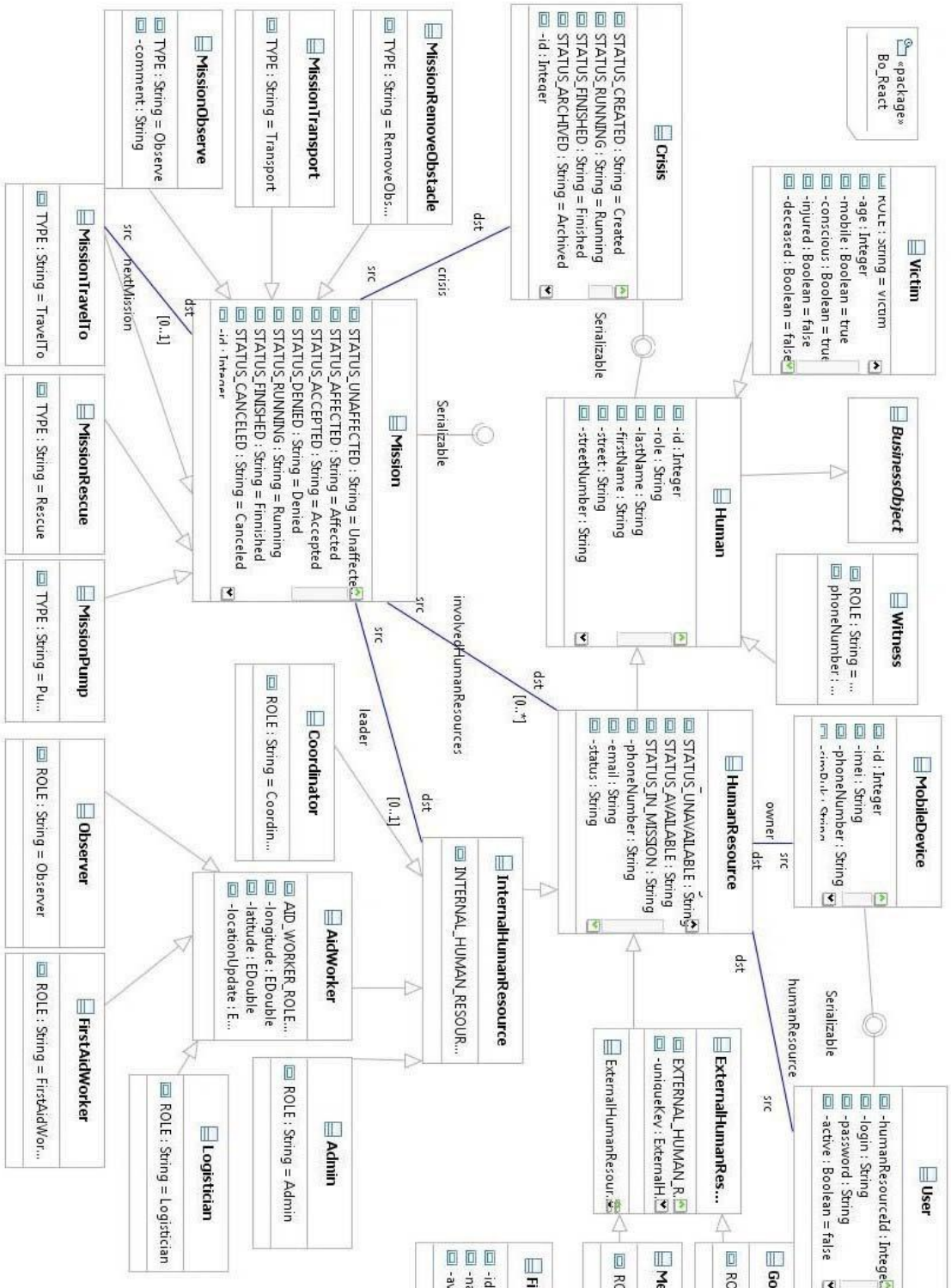
[ZJH031] T. Ziadi, L. Hélouët, J.-M. Jézéquel. (2003) Towards a UML profile for software product lines. In Proceedings of InternatioDernière mise à jour le 1nal Workshop on Product Family Engineering (PFE-5), Seana / Italy, 2003, Springer LNCS 3014, pp 129-139.

[Zsc09] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, Uirá Kulesza. (2009) VML* – A Family of Languages for Variability Management in Software Product Lines. In Proc. 2nd Int'l Conf. on Software Language Engineering (SLE'09).

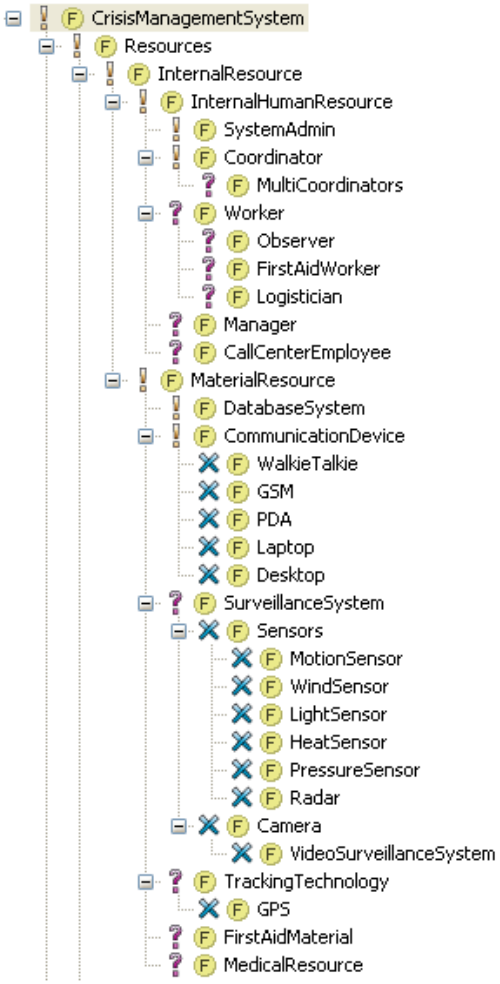
[Zsc091] Steffen Zschaler. (2009) VML*: A Generative Infrastructure for Variability Management Languages.

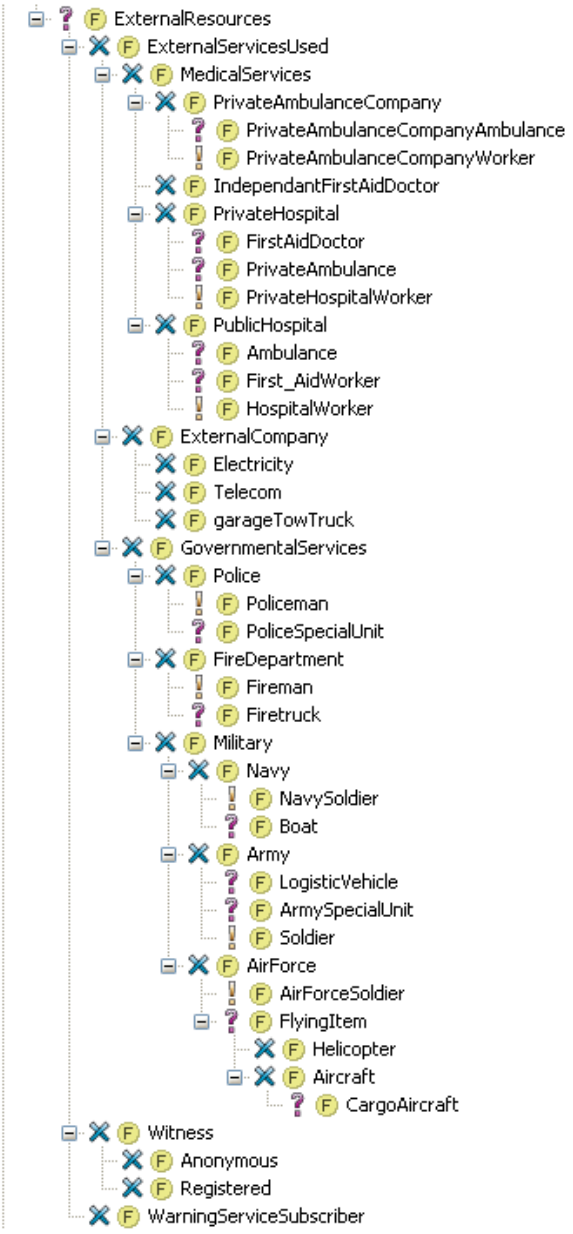
8 Annexes

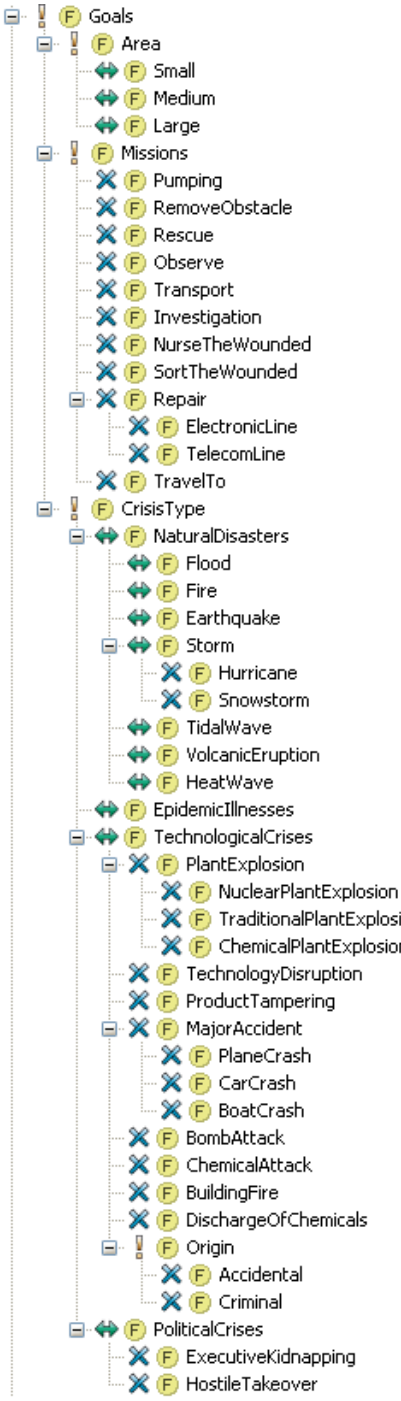
Annex A: REACT Business Objects class diagram

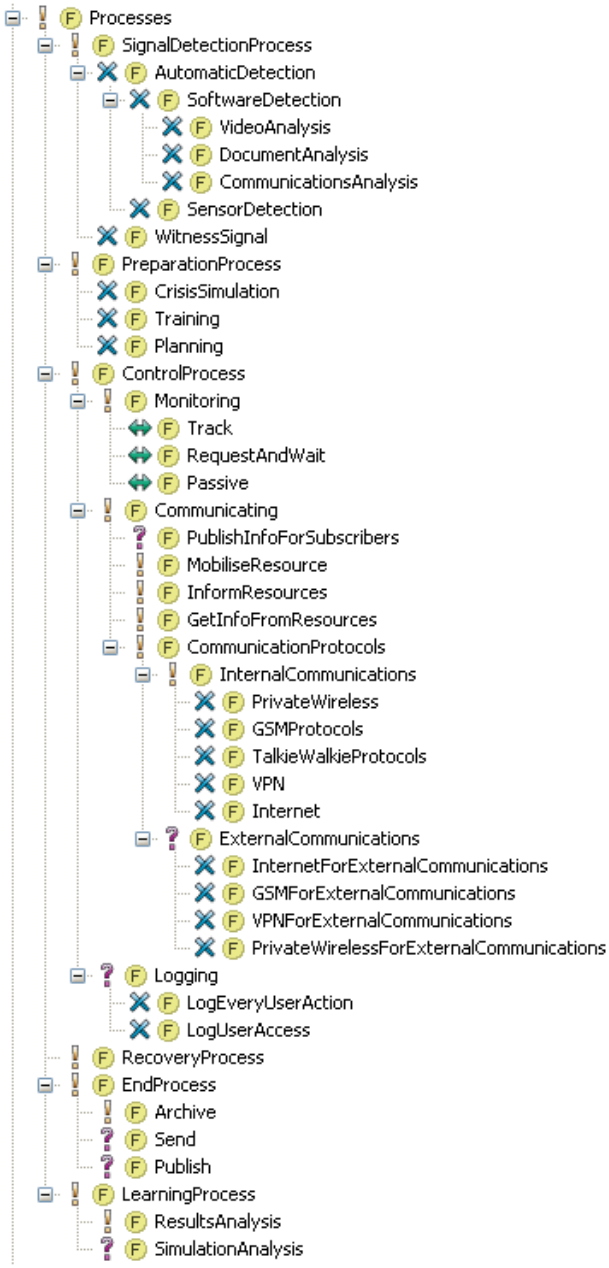


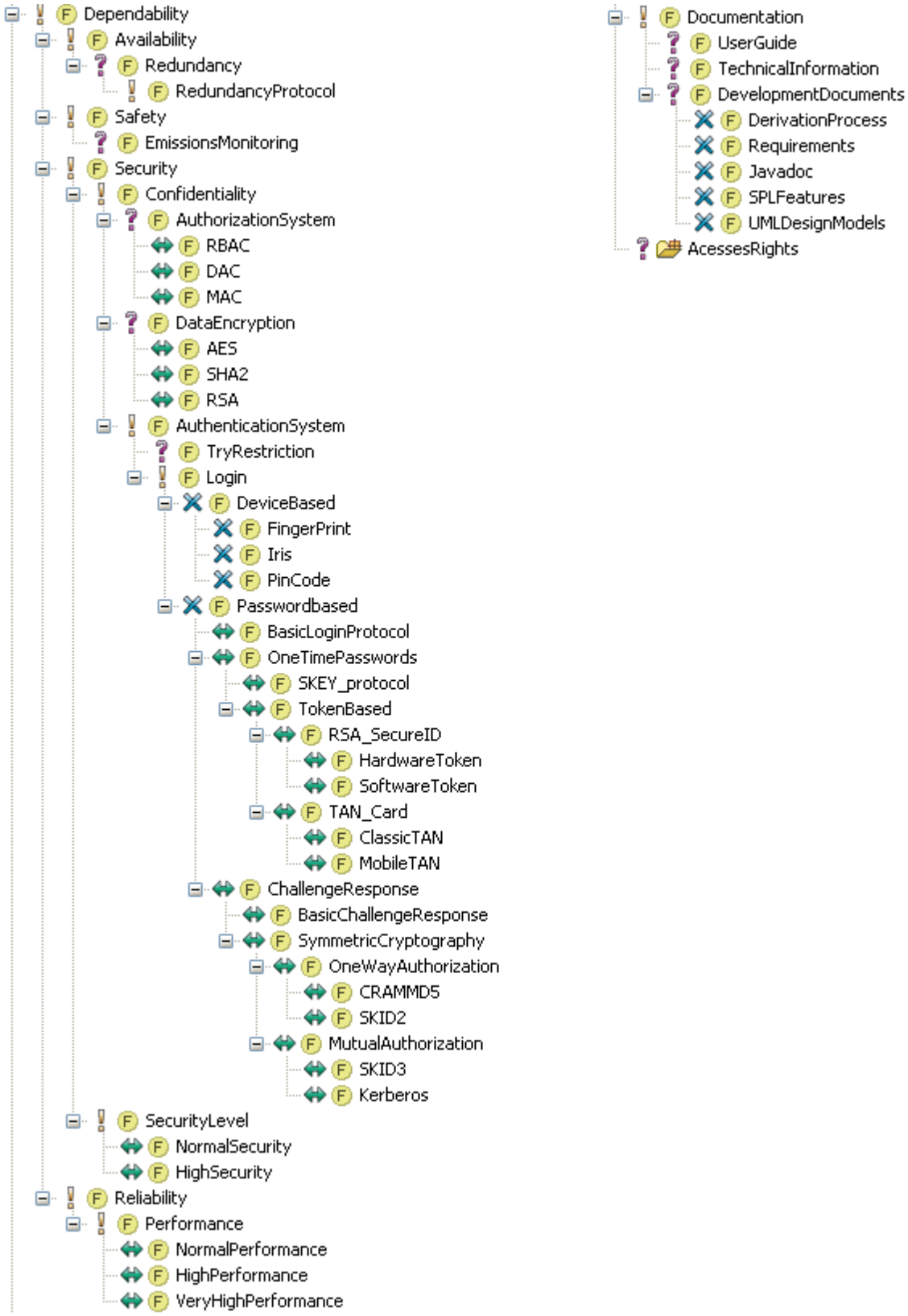
Annex B: React feature diagram






Goals
Area
Small
Medium
Large
Missions
Pumping
RemoveObstacle
Rescue
Observe
Transport
Investigation
NurseTheWounded
SortTheWounded
Repair
ElectronicLine
TelecomLine
TravelTo
CrisisType
NaturalDisasters
Flood
Fire
Earthquake
Storm
Hurricane
Snowstorm
TidalWave
VolcanicEruption
HeatWave
EpidemicIllnesses
TechnologicalCrises
PlantExplosion
NuclearPlantExplosion
TraditionalPlantExplosion
ChemicalPlantExplosion
TechnologyDisruption
ProductTampering
MajorAccident
PlaneCrash
CarCrash
BoatCrash
BombAttack
ChemicalAttack
BuildingFire
DischargeOfChemicals
Origin
Accidental
Criminal
PoliticalCrises
ExecutiveKidnapping
HostileTakeover


Processes
SignalDetectionProcess
AutomaticDetection
SoftwareDetection
VideoAnalysis
DocumentAnalysis
CommunicationsAnalysis
SensorDetection
WitnessSignal
PreparationProcess
CrisisSimulation
Training
Planning
ControlProcess
Monitoring
Track
RequestAndWait
Passive
Communicating
PublishInfoForSubscribers
MobiliseResource
InformResources
GetInfoFromResources
CommunicationProtocols
InternalCommunications
PrivateWireless
GSMProtocols
TalkieWalkieProtocols
VPN
Internet
ExternalCommunications
InternetForExternalCommunications
GSMForExternalCommunications
VPNForExternalCommunications
PrivateWirelessForExternalCommunications
Logging
LogEveryUserAction
LogUserAccess
RecoveryProcess
EndProcess
Archive
Send
Publish
LearningProcess
ResultsAnalysis
SimulationAnalysis


Dependability
Availability
Redundancy
RedundancyProtocol
Safety
EmissionsMonitoring
Security
Confidentiality
AuthorizationSystem
RBAC
DAC
MAC
DataEncryption
AES
SHA2
RSA
AuthenticationSystem
TryRestriction
Login
DeviceBased
FingerPrint
Iris
PinCode
Passwordbased
BasicLoginProtocol
OneTimePasswords
SKEY_protocol
TokenBased
RSA_SecureID
HardwareToken
SoftwareToken
TAN_Card
ClassicTAN
MobileTAN
ChallengeResponse
BasicChallengeResponse
SymmetricCryptography
OneWayAuthorization
CRAMMD5
SKID2
MutualAuthorization
SKID3
Kerberos
SecurityLevel
NormalSecurity
HighSecurity
Reliability
Performance
NormalPerformance
HighPerformance
VeryHighPerformance
Documentation
UserGuide
TechnicalInformation
DevelopmentDocuments
DerivationProcess
Requirements
Javadoc
SPLFeatures
UMLDesignModels
AcessesRights

Annex C: LASSY internship report

Rapport de stage

«Model-Driven development of Crisis Management Applications in a Software Product Line »

Leclercq Marc

Faculté d'Informatique

FUNDP

En collaboration avec la faculté des Sciences, Technologies et Communications de l'université du Luxembourg

Février 2011

Promoteurs : Patrick Heymans, Gilles Perrouin, Nicolas Guelfi, Benoit Ries

Table des matières

Introduction

Ce document représente le rapport du stage intitulé “ModelDriven development of crisis Management Applications in a Software Product Line” qui a été effectué du 15 septembre 2010 au 15 janvier 2011 au laboratoire de logiciels systèmes avancés (« Labratory for Advanced Software Systems » LASSY) de l'université du Luxembourg, en partenariat avec l'université de Namur. Sur place, j'ai été encadré par messieurs Nicolas Guelfi et Benoît Ries, tandis que messieurs Patrick Heymans et Gilles Perrouin ont suivi mon avancement depuis Namur.

Ce rapport présente un résumé des activités poursuivies, leurs motivations et leurs problématiques, sans vouloir entrer dans les détails. Un rapport de stage plus « technique » comportant les détails du stage a été rédigé pour le LASSY.

Au LASSY, j'ai travaillé sur le project REACT, dont le but est de développer une ligne de produits de management de crises. La ligne de produit de REACT est composée d'une plateforme logicilelle qui fourni des composants réutilisables, qui seront chacun utilisés dans certains produits dérivés de la plateforme, qui auront pour but de gérer le management d'une crise ou d'un type de crise bien particulier(ière), comme par exemple les tremblements de terre, les accidents routiers ou encore les explosions de centrales nucléaires. Le but de ces logiciels est d'aider à gérer les différentes ressources requises lors de la gestion de crises et de faciliter la communication entre les différents acteurs impliqués. Le sujet de stage était d'utiliser diverses techniques et outils de “modeldriven development” afin d'améliorer le processus de dérivation de produits à partir de la plateforme qui existait déjà.

La suite de ce document présente d'abord les objectifs du stage, sa problématique et ses motivations. Ensuite, la seconde partie abordera le déroulement du stage, ainsi que les activités poursuivies lors de ce stage. Enfin, la dernière section concluera ce rapport.

Objectifs du stage

Le stage avait pour but d'appliquer sur le projet REACT des techniques et outils de “model driven development” afin d'améliorer le processus de dérivation de produits existants. Pour ce faire, les objectifs annoncés du stage se déclinaient en 4 sousobjectifs:

- Modélisation de la ligne de produit REACT avec Pure::Variants. Niveau requirements: spécification des features et de leurs interdépendances. Niveau design: lien entre les features et les composants d'implémentation existants de la plateforme REACT
- Extension et intégration du modèle de la ligne de produit comme input pour le processus existant de dérivation automatique.
- Application du processus de dérivation dans le cadre de l'application de gestion de car crash.
- Modélisation exploratoire du comportement de la plateforme REACT en utilisant le langage défini par Paul Istoan.

En effet, le processus de dérivation de produits à partir de REACT, tel qui'il était avant le début stage, nécessitait premièrement de choisir soi-même les différents artefacts fournis par la plateforme (classes java, interfaces, ...) sans aucun outil, puis de créer un diagramme de classe UML de ces artefacts. Ensuite, il fallait décider des nouveaux artefacts à implémenter qui seraient spécifiques à l'application à dériver, et de nouveau en générer un diagramme de classe UML. Après cela, les 2 diagrammes sont “fusionnés” et l'application finale générée à partir de là, via une application implémentée via Kermeta , qui est un langage de métamodélisation , et KET (Kermeta Emitter Template) qui est un langage qui permet de créer des scriptes de génération de texte.

Les inconvénients de ce processus sont que l'utilisateur n'est pas du tout aidé dans le choix des artefacts, et la génération des diagrammes. Le but du stage est donc premièrement de pallier à ces inconvénients. Le fait d'utiliser un outil comme Pure::Variants permettra d'améliorer le processus de dérivation en automatisant certaines tâches telles que la génération des diagrammes de classes ou de fichiers de configuration. Aussi, il va permettre à l'utilisateur de baser sa sélection d'artefacts non plus directement à partir des artefacts, mais à partir de features contenues dans un diagramme de features, qui donnent une vue plus haut niveau et plus compréhensible de ce qu'offre la plateforme REACT, et qui seront donc liées avec les composants de la plateforme REACT qu'ils représentent.

Aussi, cette vue permettra de spécifier des contraintes, des relations entre features telles que l'exclusion mutuelle de features ou le fait qu'une feature en nécessite une autre. De plus, l'outil permet ensuite de vérifier que ces contraintes et relations sont respectées lors d'une sélection de features. Donc, en plus d'apporter une vue plus compréhensible et contenant plus d'informations, d'accélérer le processus en rendant certaines tâches automatiques (au final, l'utilisateur n'aura ici plus qu'à créer une sélection de features via un arbre reprenant les features, et où il faut juste cocher la feature), cela apportera au processus de dérivation un mécanisme qui permet d'éviter des constructions de programmes non valides, ce qui est donc une sécurité supplémentaire.

Le deuxième sous-objectif est d'intégrer la modélisation dans le processus de dérivation existant, c'est-à-dire qu'après avoir spécifié les features et leurs liens, on a besoin de pouvoir générer un diagramme de classe UML des composants d'implémentation liés aux features sélectionnées pour un produit, afin de l'utiliser comme input du processus implementé avec Kermeta et KET.

Le troisième sousobjectif, quant à lui, a pour but de tester la validité du processus défini, alors que le dernier sous objectif a pour but de représenter, avec le langage défini par Paul Istoan (un des chercheurs du LASSY), le comportement de la plateforme.

Le LASSY étant un laboratoire de recherche universitaire, un autre objectif de ce stage était, en utilisant différentes techniques de model driven development sur la plateforme REACT, de prouver que ces techniques sont utiles dans le cadre du développement d'une ligne de produit et de ces applications, ainsi que de comparer les techniques utilisées par rapport à d'autres techniques existantes dans le même domaine et de donner leurs avantages et inconvénients.

Déroulement du stage

Le stage s'est déroulé du 15 septembre 2010 au 15 janvier 2011. Durant le stage, une réunion était programmée chaque lundi avec Benoît Ries afin de présenter l'avancement de la semaine et de convenir mutuellement des futures tâches à faire. En effet, il m'a été demandé de m'approprier personnellement le sujet du stage, d'avoir ma propre opinion à ce sujet afin de pouvoir proposer moi-même certaines tâches intéressantes,... . Par exemple, alors que le sujet du stage était basé sur l'utilisation de Pure::Variants, j'ai personnellement proposé, après quelques semaines de recherches et de lecture de documentation, d'utiliser le logiciel FeatureMapper, qui est capable de s'intégrer dans Pure::Variants, car il proposait un système efficace pour générer automatiquement des diagrammes UML à partir d'une sélection de features réalisées dans Pure::Variants, ainsi qu'une interface facilement utilisable et offrant différentes vues du mapping pour aider l'utilisateur.

J'ai également participé à deux réunions avec Nicolas Guelfi, le directeur du LASSY. La première a eu pour but de dresser les grandes lignes du déroulement du stage et des activités à poursuivre, tandis que la deuxième a eu lieu peu après la moitié du stage et a eu pour but de présenter mon avancement à ce moment-là.

Aussi, pendant cette seconde réunion, on a convenu mutuellement de changer le dernier objectif du stage, qui était d'étudier la plateforme REACT d'un point de vue comportemental, en la réalisation d'une recherche sur les différentes méthodes de variability mapping dont FeatureMapper fait partie (le variability mapping étant défini comme le fait d'associer des éléments de diagrammes , par exemple des classes de diagramme de classe à des features d'un diagramme de feature). Ceci a été convenu car cela correspondait plus au sujet de mon mémoire (qui se basera en effet sur les travaux effectués au LASSY), et car il m'intéraissait plus au niveau personnel. C'est en fait la seule fois où les objectifs du stage ont été modifiés.

Aussi, un rapport de stage a été envoyé environ toutes les 2 semaines à messieurs Heymans et Perrouin afin qu'ils puissent suivre mon avancement au LASSY.

Parallèlement aux activités poursuivies (voir ci-après), il m'a aussi été demandé d'écrire un rapport de stage pour le LASSY, comprenant les activités poursuivies, leurs problématiques et motivations, ainsi que le détail de leur résultats, afin de posséder une trace écrite des résultats de ce stage, qui pourra être réutilisée par après dans le projet REACT.

Enfin, le 11 janvier,j'ai également effectué une présentation des résultats de mon stage sous forme de “staff seminar” devant les membres du LASSY (la présentation s'est donc faite en anglais, étant donné que certains de ces membres ne sont pas francophones).

Activités poursuivies

Activités préalables

Pour commencer le stage, afin de bien comprendre les notions des domaines du Software Product Line Engineering, du management de crises et de la modélisation en diagramme de features, j'ai effectué plusieurs lectures de documents à ces sujets. J'ai également lu les documents existants sur la plateforme REACT, ainsi qu'exploré le code source de la plateforme, afin de bien comprendre son fonctionnement et ses buts. Enfin, j'ai aussi effectué plusieurs tests sur le logiciel Pure::Variants pour m'y habituer.

Analyse de Pure::Variants

Avant d'utiliser le logiciel Pure::Variants pour la modélisation de la plateforme REACT, il était impératif que je possède une connaissance approfondie du logiciel et des possibilités qu'il offre. J'ai donc effectué une analyse de la syntaxe et de la sémantique des langages de feature model, de family model (utilisé pour représenter les “artefacts” comme les composants de la plateforme REACT), des variant description model (utilisés pour créer des sélections de features) et des variant result model (utilisés pour représenter les produits finis, après la transformation effectuée par le logiciel, qui est spécifiée par l'utilisateur). Aussi, sur base d'un rapport de Paul Istoan, qui avait comparé plusieurs langages de modélisation par features, j'ai comparé le langage proposé par Pure::Variants avec les langages étudiés dans ce rapport-là. Une des conclusions que j'ai établi est que le langage de feature de Pure::Variants permet d'exprimer presque tous les concepts vus dans tous les autres langages, ce qui a donc prouvé que le choix de Pure::Variants pour la modélisation (choix qui à été fait avant le stage par les promoteurs au LASSY, et qui m'a été “imposé”) était un bon choix.

Analyse de FeatureMapper

Après avoir proposé d'utiliser FeatureMapper pour créer des diagrammes UML et ainsi faire le lien entre la modélisation et le processus de drivation existant, et que cette proposition ait été acceptée par les promoteurs du LASSY, il état donc nécessaire que, à l'instar del'analyse de Pure::Variants, j'analyse le logiciel FeatureMapper. J'ai donc de nouveau étudié la syntaxe et sémantique de ce langage pour comprendre quelles constructions étaient possibles, et ce qu'elles signifiaient. De plus, mais seulement après avoir terminé l'étude de cas sur REACT (voir ciaprès), j'ai proposé de faire une recherche sur le domaine du variability mapping (c'estàdire le fait de lier des éléments de modèle à des features d'un modèe de feature) dont FeatureMapper fait partie, car c'était à la fois intéressant pour mon mémoire, et aussi pour le rapport du stage pour le LASSY, car cela a permis de comparer FeatureMapper à d'autres techniques utilisées dans ce domaine. Cette tâche a en fait remplacé celle de modélisation exploratoire de la plateforme REACT.

Etude de cas sur REACT

Après avoir fait l'étude des différents logiciels a utiliser, j'ai pu les appliquer sur REACT, afin d'une part d'améliorer le processus de dérivation de cette plateforme, et d'autre part de tester leur utilité sur cette étude de cas.

La première activité était de modéliser la ligne de produit REACT, en spécifiant les features de la ligne de produit, les liens entre ces features et leurs liens avec les composants de la plateforme.

Pour ce faire, j'ai tout d'abord caractérisé le "scope" de la ligne de produit REACT, c'est-à-dire quels produits veut-on pouvoir dériver à partir de la plateforme, et quels produits n'appartiennent pas à cet ensemble. Pour ce faire, je me suis basé sur les différents documents de requirements de REACT déjà existants, sur divers sources à propos des logiciels de gestion crises, à partir des réunions que j'ai eues avec Benoît Ries et aussi à partir du diagramme de feature existant. Le but de cette caractérisation était de bien déterminer la ligne de produit afin d'ensuite pouvoir créer une modélisation valide et complète.

Ensuite, j'ai effectué une analyse critique de la modélisation existante, qui était constituée d'un diagramme de feature existant. J'ai d'abord effectué des recherches sur les bonnes pratiques de modélisation par features, puis je les ai appliquées, ainsi que mes résultats de la détermination du scope, pour créer, à partir de ce diagramme, un nouveau diagramme de feature modélisé dans le langage de feature de Pure::Variants, dans le but de faire disparaître certains défauts de cet ancien modèles, de le rendre plus "user-friendly", plus en adéquation avec la plateforme (en effet, la plateforme avait assez bien évolué depuis la dernière version de ce diagramme), de pouvoir exprimer les relations entre les différentes features, et de pouvoir l'intégrer dans l'outil Pure::Variants pour ensuite utiliser ses capacités de transformation de modèles, de création de sélection de feature assez simple pour représenter des variantes de produits,....

Après cela, comme énoncé dans les objectifs, il fallait que j'établisse le lien entre la modélisation et les composants de la plateforme. Pour cela, après l'avoir proposé et que ça ait été accepté, j'ai utilisé le logiciel FeatureMapper, qui peut s'intégrer à Pure::Variants, et qui permet de transformer des entre autres des modèles UML en fonction d'un "variant description model" de Pure::Variants, qui modélise une sélection de feature pour une variante. J'ai choisi d'utiliser ce logiciel car ses fonctionnalités correspondaient parfaitement avec l'objectif de "mapper" des features à des éléments d'un modèle, qui représente ici l'architecture de la plateforme REACT, et en même temps avec celui d'intégrer la modélisation dans le processus de dérivation existant, qui prend en effet des diagrammes de classe UML comme input. De plus, le logiciel est assez simple d'utilisation, repose entièrement sur une interface graphique (pas de script à éditer,...) alors que ce n'est pas le cas de la solution que propose Pure::Variants pour ce problème. En effet, si on avait voulu utiliser Pure::Variants uniquement, il aurait fallu implémenter tout le processus de génération de diagramme UML. Une fois ces liens définis avec FeatureMapper, on possédait donc un processus complet de dérivation de produit, qui demande à l'utilisateur de créer des variantes de la ligne de produit en sélectionnant des features, et qui ensuite générait automatiquement un diagramme de classe UML, qui était dès lors utilisé pour générer automatiquement le produit fini. Ce processus est donc plus facile à utiliser que le précédent, plus "automatique", et permet aussi d'éviter de faire des erreurs lors de la conception, car l'outil pure::Variants comprend un "model checker" qui permet de vérifier que toutes les contraintes définies sont respectées. C'est aussi plus facile de définir un

produit à partir de features, qui représentent des concepts bien déterminés, que de se baser directement sur les composants de la plateforme (code source,...).

Par ailleurs, on a décidé d'ajouter au processus la possibilité de générer un fichier de droit d'utilisateurs, afin d'améliorer encore ce processus et aussi de tester la génération de fichier avec Pure::Variants, qui n'est pas possible de faire avec Featuremapper. Pour ce faire, j'ai d'abord créé un modèle de description de la structure XML du fichier à générer. Ensuite, j'ai ajouté à la modélisation les notions de droits d'utilisateurs, via de nouvelles features et éléments de family models. Ensuite, j'ai utilisé le module de transformation par scripts XSLT fourni par Pure::Variants. Pure::Variants permettait aussi d'utiliser une autre transformation pour cela, la transformation textuelle à base de tags sur un texte complet qui permettaient de supprimer les parties du texte non désirée. Mais cette solution était évidemment beaucoup moins efficace que le script XSLT, car elle nécessite de créer le texte entier (ici un fichier XML de tous les droits pour tous les utilisateurs) et qui doit être modifiée à chaque ajout de droit ou d'utilisateur, tandis que le script XSLT permet d'éviter cela car il effectue une analyse de la modélisation des droits d'utilisateurs et crée le fichier en fonction de cela.

Enfin, après avoir implémenté tout le processus, je l'ai testé en créant une variante destinée aux car crashs. Le test a montré que les diagrammes de classes étaient correctement générés, et ce d'une manière assez simple. Cependant, deux problèmes sont arrivés lors du test: d'une part, on s'est aperçu que FeatureMapper gérait les références vers les éléments des modèles qu'il transforme de manière absolue, ce qui nécessite soit de ne travailler que sur un poste de travail pour tout le projet, soit de retoucher le mapping pour chaque poste de travail différent si on travaille sur plusieurs postes de travail avec un outil de versionning comme SVN. D'autre part, on s'est aussi aperçu que le processus de dérivation existant développé en Kermeta (développé par un ancien stagiraire) n'était pas tout à fait fini par rapport à ces spécifications : en effet, il n'était pas encore capable de prendre en input des diagrammes de classe, mais prenait seulement un fichier texte décrivant les différents composants utilisés. Nous n'avons donc pas su tester la génération du produit fini à partir du diagramme UML, mais les composants du diagramme UML généré correspondaient aux composants qui avaient été sélectionnés lors des tests de ce processus de dérivation, ce qui laisse supposer que le nouveau processus s'intègre correctement avec l'ancien.

Conclusion

Ce rapport a présenté le stage que j'ai effectué au LASSY du 15 septembre 2010 au 15 janvier 2011. Premièrement, j'ai introduit les objectifs du stage tels qu'ils ont été définis au début de celui-ci. Ensuite, j'ai relaté le déroulement du stage et le fait qu'un des objectifs du stage a été changé en cours de route. Enfin, j'ai expliqué quelles activités j'ai poursuivies lors de ce stage.

Nous avons vu que le stage a permis d'améliorer le processus de dérivation de produits à partir de la plateforme REACT, et ce grâce à l'application des outils Pure::Variants et FeatureMapper. Ils ont en effet permis d'automatiser la plupart des tâches, l'utilisateur n'ayant plus qu'a spécifier une variante de la ligne de produit via une sélection de feature. De plus, ces outils apportent la possibilité d'exprimer de nouveaux concepts absents dans l'ancien processus, telle la possibilité de spécifier des contraintes, et aussi de vérifier ces contraintes pour chaque produit, ce qui réduit la probabilité de construire des produits invalides. Enfin, ils apportent des méthodes de visualisation utiles pour l'utilisateur.

De plus, le stage a permis d'effectuer une étude de cas de l'application de certains outils et techniques, de les comparer par rapport à d'autres outils de leur domaine,et aussi d'en donner certains avantages et inconvénients par rapport à notre étude de cas ou par rapport à une comparaison.

D'un point de vue plus personnel, le stage m'a tout d'abord permis d'acquérir de nouvelles connaissances dans les domaines du “software product line engineering”. Mais il m'a aussi donné un aperçu de ce qu'est le travail de chercheur dans un laboratoire universitaire, et d'une manière plus générale un aperçu du travail dans la recherche. Enfin, ce stage, ainsi que le rapport de stage technique que j'ai réalisé au LASSY, me seront utiles pour la rédaction de mon mémoire.